# Elastic Machine Learning Algorithms in Amazon SageMaker

Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, Alex Smola

Amazon AI

{libertye,zkarnin,bxiang,rouesne,barisco,rnallapa,juliod,sadoughi,yastasho,pialidas,balioglu,saswatac, madhavjh,gautierp,arpin,tjnsch,flunkert,yuyawang,gasthaus,stellalo,rangapur,dsalina,sseb,smola}@amazon.com

## ABSTRACT

There is a large body of research on scalable machine learning (ML). Nevertheless, training ML models on large, continuously evolving datasets is still a difficult and costly undertaking for many companies and institutions. We discuss such challenges and derive requirements for an industrial-scale ML platform. Next, we describe the computational model behind Amazon *SageMaker* which is designed to meet such challenges. *SageMaker* is an ML platform provided as part of Amazon Web Services (AWS), and supports incremental training, resumable and elastic learning as well as automatic hyperparameter optimization. We detail how to adapt several popular ML algorithms to its computational model. Finally, we present an experimental evaluation on large datasets, comparing *SageMaker* to several scalable, JVM-based implementations of ML algorithms, which we significantly outperform with regard to computation time and cost.

**ACM Reference Format:**

Liberty et al.. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3318464.3386126



*SIGMOD'20, June 14–19, 2020, Portland, OR, USA*



ACM ISBN 978-1-4503-6735-6/20/06…$15.00

https://doi.org/10.1145/3318464.3386126

## 1 INTRODUCTION

Machine learning (ML) has become an integral part of modern software systems. Unfortunately, training ML models on large, continuously evolving datasets is still a significant undertaking for many companies and institutions, especially if ML is not their core competency. Building an industrial-scale model training platform for such cases involves a set of challenges, many of which are not addressed by current systems available in academia and open source.

(*i*) Support for *incremental training and model freshness:* It is highly uncommon to encounter large static datasets. In most cases, data keeps being generated constantly, which is often addressed with an unwelcome trade-off between training cost and accuracy. Training on a large subset of the data produces accurate models but can become extremely costly and slow, while training on new, small updates of the data (e.g., the last day) is cheaper but might not lead to very accurate results. Therefore, industrial ML platforms have to support incremental model training to regularly and cost-efficiently update existing models and to quickly provide accurate and 'fresh' models.

(*ii*) *Predictability of training costs:* for large amounts of data, customers need to be able to roughly estimate in advance how much a training job would cost and how long it would run for. It is difficult to estimate the cost ahead of time for many scalable systems, which do not support incremental learning with linear update times or have irregular performance drops for high-dimensional models [5].

(*iii*) *Elasticity and support for pausing and resuming training jobs:* large-scale ML scenarios often result in imbalanced workloads, where data scientists spend several days without running a single job (while they are collecting data or writing code) and then they launch several large concurrent training jobs on hundreds of machines. They also may want to pause and resume such jobs, e.g., for hyperparameter tuning or if

their compute bandwidth has irregular limitations. A cloud ML platform should reduce the operational complexity of such scenarios.

(*iv*) Furthermore, an industrial-scale ML platform must be able to handle *ephemeral data*, as some data streams like network traffic or video streams are never meant to be stored, which makes ingesting and learning on such data challenging.

(*v*) Finally, an ML platform must automate *hyperparameter optimization and model tuning* as much as possible. These tasks are already tedious even for small use cases, let alone, for large scale machine learning scenarios. Automating this part of the platform is crucial for driving down training costs and supporting non-ML expert users.

In this paper, we present the computational model (Section 2) and algorithms (Section 3) of Amazon *SageMaker*, a system for scalable, elastic model training on large streams of data, available as part of Amazon's cloud offerings.[1] We choose a streaming model to meet requirements like linear update time, pause/resume capabilities and elasticity. Operating in such a constrained setting is mathematically complex. For example, consider the task of computing the median of a stream of numbers. The best possible approximation for finding quantiles (e.g., the median) in streams was only recently discovered [18], and the resulting algorithm is significantly more complex than offline approaches [14]. We argue, however, that the advantages of a streaming model are so significant that they are well worth the additional complexity: The cost of processing additional data is independent of the size of the input data, the memory footprint of the algorithms is fixed, and ideally, such a solution exhibits close-to-linear scalability for both runtime and compute costs of model training.

We outline some design choices for *SageMaker*'s implementation in Section 4, refering to its use of *Apache MXNet* [10] and the maintenance of shared state via a *parameter server* [21]. We experimentally evaluate our system against popular JVM-based solutions for scalable model training in Section 6. In summary, this paper provides the following contributions:

- We discuss challenges and requirements for building an industrial-scale elastic ML platform (Section 1).
- We describe the computational model (Section 2) and selected algorithms of *SageMaker*, which support incremental, resumable and elastic learning, as well as automatic hyperparameter optimization (Sections 3 & 4).
- We compare *SageMaker* to several scalable, JVM-based algorithm implementations (Section 6), which we significantly outperform with regard to computation time and cost.

[1] https://aws.amazon.com/sagemaker

## 2 COMPUTATIONAL MODEL

*SageMaker* assumes distributed streaming data and a shared model state. The combination of a streaming model and shared state enables a convenient and simple abstraction for distributed learning algorithms. Our runtime is in charge of managing the state including weight updates, synchronization, reading inputs, writing outputs, logging, reporting metrics, containerization, serialization, deserialization, and many other tasks.

As a result, large scale learning algorithms can be integrated by implementing only three functions, namely, `initialize`, `update`, and `finalize`. (*i*) `initialize`: sets up an initial (often empty) state such that training can begin. (*ii*) `update`: receives a data stream and the state, and updates the state accordingly. At the system level, the `update` function is simultaneously executed on many machines in parallel, and must be implemented such that the updates on partitions of the input stream can be merged together. (*iii*) `finalize`: receives the state (and potentially configuration parameters) and outputs the final result of the computation (typically an ML model).[2] This abstraction allows one to optimize a wide of variety machine learning objective functions by writing only three simple functions while still enjoying all the benefits of a massively distributed, streaming, production-ready, model training platform.

**Example**. We illustrate our computational model via the example algorithm of computing the median of a stream of numbers in a distributed fashion. Recall that the median of a set of points $x_1, \ldots, x_n$ is equal to $\operatorname{argmin}_z \frac{1}{n} \sum_{i=1}^{n} |x_i - z|$. Applying stochastic gradient descent (SGD) to minimize this convex function gives Algorithm 1: We maintain a guess for the median, and update this guess after every observed item (we increment our guess if it is smaller than the observed item and decrement it otherwise).

The `initialize` function (Line 1) initializes shared variables in the global state object. (Note that this function will not be called when we resume a previously paused job, as there is already an existing state). The `update` function (Line 4) updates the state in response to a stream of mini-batches of data. In our example, we constantly update the state (i.e., the estimated median) at the end of each batch (Line 14), as well as the number of observed data items (Line 13). Note that the `state.push` operation does not override shared variables in the state. Instead, it invokes a server-side aggregation function, which adds the pushed value to the global value in this case. Our runtime ensures that the `finalize` function (Line 16) is only called after all workers have finished iterating over the data. Thereby, we ensure that the result of the `state.pull` invocation (Line 17)

[2] This computational model is similar to algebraic abstractions for distributed aggregation functions in parallel databases [9, 38].

**Algorithm 1:** Distributed computation of the median via SGD.

```
1  function initialize(state):
2      state.initialize('median', 0) \\ first guess is 0
3      state.initialize('n',0) \\ number of items seen
4  function update(state, data_stream, synchronized):
5      foreach mini_batch in data_stream do
6          current_median = state.pull('median')
7          n = state.pull('n')
8          inc = 0
9          batch_size = 0
10         foreach item in mini_batch do
11             inc += 1 if item > current_median else −1
12             batch_size += 1
13         state.push('n', batch_size)
14         state.push('median', inc / sqrt(n + batch_size))
15         if synchronized then state.barrier()
16 function finalize(state):
17     return state.pull('median')
```

reflects all updates. Our system additionally supports a `barrier` function on the state object (Line 15), to allow the algorithm designer to control the synchronicity of the computation (e.g., to enable asynchronous variants of SGD).

## 3 ALGORITHMS

While deep learning approaches [1, 10, 33] have become increasingly popular in recent years, there is still a huge demand for classic, well-understood ML algorithms, which drive mission-critical ML systems like click prediction in online advertising [29]. We choose algorithms[3] that are widely used and adapt well to a streaming scenario:

**Linear Learner**. The linear learner algorithm is designed to solve regression or classification problems. We learn a corresponding linear model via stochastic gradient descent. A special feature of our algorithm design is its support for the simultaneous exploration of different training objectives by training multiple models in parallel.

**Factorization Machines**. A factorization machine [23, 35, 36] is a general-purpose supervised learning algorithm for both classification and regression tasks. It is an extension of a linear model that is designed to capture interactions between features within high dimensional sparse datasets parsimoniously. We learn factorization machines via SGD for both classification and regression problems in *SageMaker*.

[3]The full list of integrated algorithms is available at https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html

**K-Means Clustering** is an unsupervised algorithm for clustering a dataset into $k$ groups. We implement a streaming variant of $k$-means, which combines ideas from stochastic/EM optimization [25, 40], coresets [11, 13] and online facility location [24, 42]. In our experience, previous streaming solutions are either impractical in terms of runtime or provide a significantly less accurate solution, compared to multi-pass solutions such as Lloyd's iteration [25] combined with k-means++ [2] or k-means|| [3] initialization.

**Principal Component Analysis (PCA)** is an unsupervised algorithm to reduce the dimensionality of a dataset while still retaining as much variance as possible. We design a deterministic version of PCA for datasets with a moderate number of observations and features, which requires $O(d^2)$ memory, with $d$ being the input dimension. Additionally, we design an approximate version that applies random projections to reduce the memory requirements to $O(kd)$, where $k$ denotes the number of required components.

**Neural Topic Model (NTM)**. Our neural topic model is an unsupervised algorithm that learns latent representations of large collections of discrete data, such as a corpus of documents. We decided to base our algorithm on the variational autoencoder [31], to achieve fast inference compared to classic alternatives, which require iterative computations as in variational inference or Gibbs sampling.

**Time Series Forecasting with DeepAR** is a probabilistic forecasting algorithm based on recurrent neural networks [12]. In contrast to other popular time series forecasting algorithms like [15], DeepAR learns a single (global) model [16] over a collection of time series and thereby addresses the class of operational forecasting problems often encountered in enterprise applications [8, 17].

## 4 IMPLEMENTATION

Customers of AWS *SageMaker* are interested in reducing the running time and dollar cost of training irrespective of the number and types of machines used under the hood. Our system is therefore designed to take advantage of multiple EC2 instance types, to support modern hardware like GPUs, and to scale out model training across many machines (Figure 1).

**Execution on Modern Hardware**. In order to operate across CPUs and GPUs seamlessly, most algorithms in *SageMaker* use the MXNet [10] library as an interface to the underlying hardware. MXNet represents ML algorithms via a computational graph of tensor operators, optimizes this graph (e.g, to re-use allocated memory), assigns the operators to devices, and efficiently executes the computation in parallel.

**Distributed Learning and State Management via a Parameter Server**. Distribution is achieved via a *parameter server* which maintains the shared state of all the machines

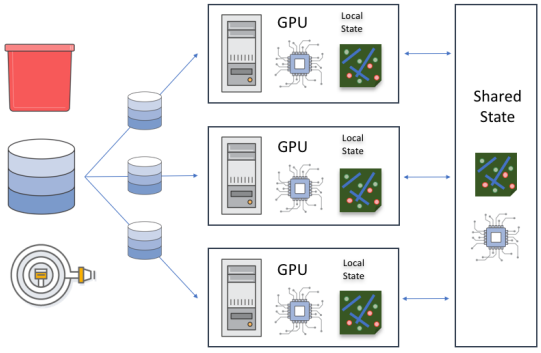



**Figure 1: System design: workers continuously learn on streaming data (using MXNet-based algorithms) while shared state is maintained via a parameter server.**

participating in training. The parameter server is designed for fast update speeds via asynchronous communication and allows us to loosen the consistency of parameter updates. Note that the tradeoff space between update consistency and convergence speed is well explored for ML workloads [21, 22, 32, 44]. We implement the shared state abstraction as well as the server-side aggregation functions required for our computational model (outlined in Section 2) via MXNet's parameter server called `KVStore` [10].

**Model Abstraction & Expressive State**. We provide a model abstraction for the outputs of training algorithms to enforce common behavior among the different ML models. A model must implement two functions: (*i*) the `score` function evaluates a model using a metric on a dataset, and is used for debugging and hyperparameter optimization (HPO). (*ii*) The function `evaluate` receives a batch of data and computes the output of the model. Note that this output is model specific, and can be a single number representing a predicted class, a vector in the case of dimensionality reduction or a complete embedding matrix.

We additionally design our algorithms to maintain an expressive state object out of which many different models can be created. Our state for $k$-means for example acts like a core set, and can be used to solve k-means for any $k \leq k_{max}$ without retraining. Similarly, $k$-nearest-neighbor models[4] (not covered in this paper) with different values for $k$ can be fitted on the corresponding state without retraining. Moreover, we apply 'model packing' [39] in the state object of the linear learner, which means that it maintains a collection of differently parameterized linear models rather than a single model.

[4]https://docs.aws.amazon.com/sagemaker/latest/dg/k-nearest-neighbors.html

**Model Tuning and Hyperparameter Optimisation**. Typical hyperparameter optimisation workloads comprise of several training jobs that are run in parallel with different hyperparameters to determine the best configuration, which can quickly become tedious and costly. The pause-resume capabilities of our streaming model enable a couple of beneficial techniques for HPO workloads. Users can for example add more data to their model every day rather than having to retrain on the history. Moreover, the combination with the ability of our models to produce early results allows for 'multi-fidelity' HPO - instead of fully training $N$ models, users can start training, say, $20N$ models, and stop those that do not perform well early-on.

## 5 RELATED WORK

Scalable machine learning has been in the focus of the academic community in the last years. Many different systems have been proposed, ranging from ML on map-reduce-like systems [7, 30], to specialised systems [26, 44] and parameter servers [21, 22, 39]. In recent years, deep learning engines, such as Tensorflow [1], MXNet [10] or Pytorch [33] have become the dominant choice.

Putting ML applications into production incurs many data management challenges [8, 19, 34, 37, 41], and recent efforts focus on systems for managing the end-to-end lifecycle in ML like the Tensorflow Extended Platform [4] or SystemDS [6].

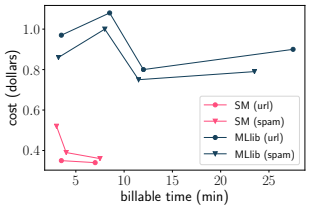
## 6 EVALUATION

We compared *SageMaker* against several JVM-based solutions for scalable machine learning. Namely, popular[5] algorithms from Apache Spark MLlib 2.2 [30] and Mallet 2.0.6 [28].

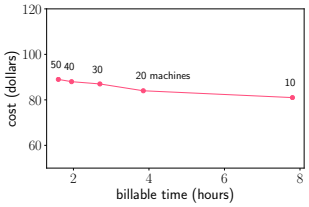
We report the AWS instance types leveraged for each experiment individually. Note that we use different instance types based on the ability of the systems to leverage modern hardware (e.g., in contrast to *SageMaker*, MLlib does not make use of GPUs). As *SageMaker* has been designed to enable scalable, cost-effective machine learning in the cloud, we evaluate the majority of our experiments with respect to the billable time and cost in dollars for solving an ML problem.

**Linear Learner**. We compare *SageMaker*'s linear learner to MLlib's logistic regression implementation on two classification tasks involving 30GB of web-spam and web-url classification data. We repeat the experiment leveraging 10, 20, 40 and 80 `m4.xlarge` instances for MLlib, and 10, 20 and 40 `m4.xlarge` instances for *SageMaker* (we stopped at 40, as *SageMaker* already provided an extremely low runtime). We ensure that both approaches achieve the same accuracy on the task, and plot the resulting computing cost and machine time in Figure 2a.

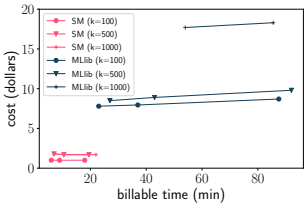
[5]https://spark.apache.org/powered-by.html



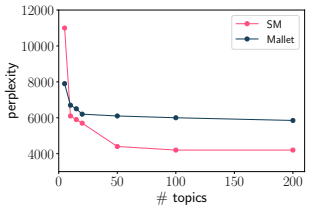

**(a) Learning a linear classifier on web spam and url data (30 GB) with varying numbers of machines.**




**(b) Scaling behavior of the factorization machines implementation on a 1TB click prediction advertising dataset.**




**(c) K-Means clustering of the *GDelt* dataset (372 GB) with varying numbers of clusters and machines.**




**(d) Perplexity scores for differently sized topic models on a news dataset with 200K documents for *SageMaker* and mallet.**

**Figure 2: Comparison of *SageMaker* against several scalable, JVM-based algorithm implementations for different ML use cases. *SageMaker* is able to train models both faster and more cost-effective in the majority of cases.**

We find that *SageMaker* is up to 8-times faster than MLlib, and can provide results two to three times cheaper when using the same amount of training time.

**Factorization Machines**. Next, we evaluate the scalability of our factorization machines implementation. We do not compare against MLlib in this setting, because it does not contain an implementation of this algorithm. We train a classifier on a 1TB click prediction dataset from the advertising domain, and evaluate a weak-scaling scenario (scaling the hardware but not the workload) using 10, 20, 30, 40 and 50 `m4.xlarge` instances.

The results are shown in Figure 2b, and we find that *SageMaker* exhibits a close-to-linear scaling behavior (indicated by a very modest increase in computation cost) when increasing the cluster size in this setting.

**K-Means**. We evaluate *SageMaker*'s k-means implementation on the GDELT news dataset [20] (372GB), and compare it against MLlib's streaming k-means implementation (based on [40]). We vary the number of clusters $k$ between 100, 500 and 1000, and leverage 4, 8 and 12 instances for the experiments. Note that we use `m4.xlarge` instances for MLlib (which does not make use of GPUs) while *SageMaker* runs on `p2.xlarge` instances, leveraging the graphics card via MXNet.

We plot the results in Figure 2c, and observe that *SageMaker*'s implementation is up to 10 times faster and significantly cheaper while achieving a sum-of-squared-differences score that is 5%-8% lower than that of MLlib, depending on the value of $k$.

Finally, we run two experiments to measure the prediction quality of our **neural topic model (NTM)** and **time series forecasting** algorithm **(DeepAR)**. We measure the perplexity of NTM for a growing number of topics on a 200K-document news dataset with 100K-word vocabulary from the New York Times corpus[6], and compare it to the perplexity achieved by the Mallet library [28] in Figure 2d. We find that NTM achieves lower perplexity scores than Mallet in all cases where the number of topics is larger than five. Next, we compare DeepAR (with default settings) to the winning solution [43] of the recent M4 forecasting competition [27].

We find that DeepAR achieves state-of-the-art performance (sMAPE of 0.12, compared to 0.114; MASE of 1.50 compared to 1.54; owa of 0.84 compared to 0.82; MSIS of 12 compared to 12.2) while exhibiting an inference speed that is comparable to conceptually much simpler R forecasting packages [15].

## 7 CONCLUSION

We discussed the challenges in building our ML platform *SageMaker*. We described its computational model, and gave an overview of how to implement several common ML algorithms with support for online learning, automatic hyperparameter optimization, and elastic learning. We experimentally evaluated our platform on large datasets and showed the performance gains achievable over existing JVM-based scalable algorithm implementations.

A limitation of specialised systems like ours is that they often have to execute preprocessing workloads on a separate system, which may reduce some of their performance benefits.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. *OSDI* (2016), 265–283.

[2] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. *SISAM* (2007), 1027–1035.

[6] https://catalog.ldc.upenn.edu/LDC2008T19

[3] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable k-means++. *VLDB* 5, 7 (2012), 622–633.

[4] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. (2017), 1387–1395.

[5] Christoph Boden, Andrea Spina, Tilmann Rabl, and Volker Markl. 2017. Benchmarking data flow systems for scalable machine learning. In *Workshop on Algorithms and Systems for MapReduce and Beyond at ACM Sigmod*.

[6] Matthias Boehm, Iulian Antonov, Mark Dokter, Robert Ginthoer, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, and Benjamin Rath. 2019. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. arXiv:cs.DB/1909.02976

[7] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *VLDB* 9, 13 (2016), 1425–1436.

[8] Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, and Yuyang Wang. 2017. Probabilistic demand forecasting at scale. *VLDB* 10, 12 (2017), 1694–1705.

[9] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A framework for integrating batch and online mapreduce computations. *VLDB* 7, 13 (2014), 1441–1451.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ML Systems workshop at NeurIPS* (2015).

[11] Dan Feldman, Melanie Schmidt, and Christian Sohler. 2013. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. *SIAM* (2013), 1434–1453.

[12] Valentin Flunkert, David Salinas, Jan Gasthaus, and Tim Januschowski. 2019. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* (2019).

[13] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. 2003. Clustering data streams: Theory and practice. *TKDE* 15, 3 (2003), 515–528.

[14] Charles AR Hoare. 1961. Algorithm 65: find. *Commun. ACM* 4, 7 (1961), 321–322.

[15] Rob J Hyndman and Yeasmin Khandakar. 2008. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software* (2008).

[16] Tim Januschowski, Jan Gasthaus, Yuyang Wang, Syama Sundar Rangapuram, and Laurent Callot. 2018. Deep Learning for Forecasting: Current Trends and Challenges. *Foresight: The International Journal of Applied Forecasting* 51 (2018), 42–47.

[17] Tim Januschowski and Stephan Kolassa. 2019. A Classification of Business Forecasting Problems. *Foresight. The Applied Forecasting Journal* (2019).

[18] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. *FOCS* (2016), 71–78.

[19] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. 2016. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record* 44, 4 (2016), 17–22.

[20] Kalev Leetaru and Philip A. Schrodt. 2013. GDELT: Global data on events, location, and tone. *ISA Annual Convention* (2013). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.686.6605

[21] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. *OSDI* (2014), 583–598. http://dl.acm.org/citation.cfm?id=2685048.2685095

[22] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *NeurIPS* (2014), 19–27.

[23] Mu Li, Ziqi Liu, Alexander J. Smola, and Yu-Xiang Wang. 2016. DiFacto: Distributed Factorization Machines. *WSDM* (2016), 377–386. https://doi.org/10.1145/2835776.2835781

[24] Edo Liberty, Ram Sriharsha, and Maxim Sviridenko. 2016. An algorithm for online k-means clustering. *Algorithm Engineering and Experiments workshop at SIAM* (2016), 81–89.

[25] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137.

[26] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB* 5, 8 (2012), 716–727.

[27] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. 2019. The M4 Competition: 100,000 time series and 61 mehotds. *International Journal of Forecasting* (2019).

[28] Andrew Kachites McCallum. 2002. MALLET: A Machine Learning for Language Toolkit. (2002). http://mallet.cs.umass.edu.

[29] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad click prediction: a view from the trenches. *KDD* (2013), 1222–1230.

[30] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.

[31] Yishu Miao, Lei Yu, and Phil Blunsom. 2016. Neural variational inference for text processing. *ICML* (2016).

[32] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. *NeurIPS* (2011), 693–701.

[33] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6 (2017).

[34] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *SIGMOD Record* 47, 2 (2018), 17–28.

[35] Steffen Rendle. 2010. Factorization Machines. *ICDM* (2010), 995–1000.

[36] Steffen Rendle. 2012. Factorization Machines with libFM. *ACM TIST* 3 (2012), 57:1–57:22.

[37] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, Gyuri Szarvas, Manasi Vartak, Samuel Madden, Hui Miao, Amol Deshpande, et al. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Engineering Bulletin* 41, 4 (2018), 5–15.

[38] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. 2019. Differential Data Quality Verification on Partitioned Data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1940–1945.

[39] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. 2014. Factorbird-a parameter server approach to distributed matrix factorization. *Distributed ML and Matrix computations workshop at NeurIPS* (2014).

[40] David Sculley. 2010. Web-scale k-means clustering. *WWW* (2010), 1177–1178.

[41] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *NeurIPS* (2015), 2503–2511.

[42] Michael Shindler, Alex Wong, and Adam W Meyerson. 2011. Fast and accurate k-means for large datasets. *NeurIPS* (2011), 2375–2383.

[43] S Smyl, J Ranganathan, and A Pasqua. 2018. M4 Forecasting Competition: Introducing a New Hybrid ES-RNN Model. *URL: https://eng. uber. com/m4-forecasting-competition* (2018).

[44] Ce Zhang and Christopher Ré. 2014. Dimmwitted: A study of main-memory statistical analytics. *VLDB* 7, 12 (2014), 1283–1294.