Probabilistic Demand Forecasting at Scale

Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski,
Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, Yuyang Wang
Amazon
{jooshenb,flunkert,gasthaus,tjnsch,langed,dsalina,sseb,matthis,yuyawang}@amazon.com

ABSTRACT

We present a platform built on large-scale, data-centric machine learning (ML) approaches, whose particular focus is *demand forecasting in retail*. At its core, this platform enables the training and application of probabilistic demand forecasting models, and provides convenient abstractions and support functionality for forecasting problems. The platform comprises of a complex end-to-end machine learning system built on Apache Spark, which includes data preprocessing, feature engineering, distributed learning, as well as evaluation, experimentation and ensembling. Furthermore, it meets the demands of a production system and scales to large catalogues containing millions of items.

We describe the challenges of building such a platform and discuss our design decisions. We detail aspects on several levels of the system, such as a set of general distributed learning schemes, our machinery for ensembling predictions, and a high-level dataflow abstraction for modeling complex ML pipelines. To the best of our knowledge, we are not aware of prior work on real-world demand forecasting systems which rivals our approach in terms of scalability.

1. INTRODUCTION

Forecasting product demand is one of the core challenges in any retail business, and essentially answers the following question: What is the probability distribution of the demand of an item for a specific time horizon starting from a date in the future? Among its many benefits, a predictive forecast is a key enabler for a better customer experience through the reduction of out-of-stock situations, and for lower costs due to better planned inventory and less write-off items.

In this paper, we describe a machine learning platform for probabilistic demand forecasting. At its core, this platform is an execution engine for state-of-the-art demand forecasting algorithms, which provides high-level abstractions for data preparation, feature engineering, distributed training and evaluation, as well as a set of tools for automating common tasks. It has been built to serve experimentation and, at the same time, meet the demands of production use cases which require forecasts for millions of items.

Algorithmic Challenges. While there exist many established methods for computing forecasts for items with a lifecycle of several years and stable promotional and seasonal effects [19], large catalogues may additionally contain items which exhibit a number of peculiarities that set them apart from the typical demand forecasting scenario. Examples include short product cycles, a large ratio of highly seasonal items, strong promotional effects, and very sparse demand at the individual item level. Tackling demand forecasting for such kinds of items requires the combination of time series methodology and machine learning methods. We adapt known techniques and combine them with a set of features produced from raw data.

System-Specific Challenges. Researchers are becoming more and more aware of the difficulties of building and maintaining complex end-to-end ML systems [25, 17]. In our experience, the main challenge is to design a system which on the one hand meets all requirements to run stably and reliably in production scenarios but is on the other hand still flexible enough to allow for rapid experimentation and algorithm development. This experimentation is typically iterative rather than one-shot, so that successful ML systems have to allow for rapid evaluation of different models and features. Existing ML workbenches, such as Matlab, R and NumPy have been designed for exactly this purpose. The downside of these platforms is that, unless special care is taken, the resulting solutions are neither scalable nor easily maintainable, and the support for distributed computations is limited. On the other hand, production-grade systems written in a compiled language, if used properly, deliver solutions that are fast, maintainable and stable. The price for this robustness is usually increased code complexity and the lack of many of the abstractions that make rapid experimentation possible. Because of this, often a hybrid approach is used: First, data scientists conduct prototyping and experimentation using interactive, high-level platforms, after which a different team of engineers re-implement their code in a production environment to create a scalable and maintainable solution for production. This works well when (a) the experiments during prototyping can easily be performed on a single machine, and (b) only iterative improvements to the system are necessary after it has left the prototyping stage. However, in large-scale machine learning problems such as demand forecasting for large catalogues, typically neither of these conditions is true: the tendency to combine simple models with large datasets to deliver accu-

Proceedings of the VLDB Endowment, Vol. 10, No. 12

rate solutions limits the effectiveness of small-scale experiments, and iterating on initial solutions is usually required as well as re-tuning of models to address model drift. Also, frequent reimplementation of experimental code incurs high overhead, is error-prone and limits the agility of the development process. Therefore, it is highly desirable to have a single system and codebase for both deployment and experimentation.

Heterogeneous Operation Modes. The combination of a unified codebase with the aforementioned challenges of a real-world use case requires systems like ours to support a variety of different *operation modes*. In general, these operation modes fall into two different categories: Firstly, *ad-hoc usages* of the system, such as conducting single machine experiments on small datasets to debug algorithms and fine-tune models, or running a single learning algorithm on a cluster to test model changes on a larger dataset. These ad-hoc runs will typically be executed several times a day by individual machine learning scientists. The main challenge here is to make these runs easy to execute and provide results fast. Secondly, there will be a set of *automated usages* of such a system, which typically leverage large clusters. The most important of these cases will obviously be runs in production scenarios which have to adhere to critical service level agreements. Typically, these runs compute forecasts for a large number of items using a predefined ensemble of learning algorithms with a fixed set of features and hyperparameters. Further examples for resource-intensive operation modes are automatic model selection workloads, which employ grid search or bayesian optimization techniques to explore large spaces of features and hyperparameters. Related to these are workloads which explore and evaluate different assignments of items to learning algorithms in order to determine well-working ensemble configurations. Finally, whenever new system versions need to be released, so-called 'backtest' workloads must be executed. These backtests compare several different ensemble configurations and software versions, and require the generation of a vast amount of pre-defined evaluations and reports.

In the remainder of this paper, we detail how we tackle the aforementioned challenges and design a system that supports all the required operation modes. We start by introducing probabilistic demand forecasting (Section 2), and afterwards give an overview of the design and implementation of our system (Section 3), which consists of loosely coupled components that exchange data via a distributed filesystem. In Section 4, we detail three technical aspects, which help us to form an end-to-end ML system from the bottom up: how to distributedly train machine learning models, how to declaratively assign items to these models, and finally how to define and execute the resulting complex ML pipelines. We close the paper with an evaluation of the presented distributed learning schemes (Section 5) and a summary of our learnings from building the platform (Section 6). In particular, we highlight the following properties of our system:

- Our modular system architecture, laying the foundation for a complex end-to-end machine learning system (Section 3).
- A set of distributed learning schemes implemented on top of a distributed dataflow engine for computing different variations of global models (over all items) and local models (over individual items) (Section 4.1).
- Machinery to produce ensemble predictions from complex model combinations (Section 4.2).
- A high-level dataflow abstraction for modeling complex ML pipelines (Section 4.3).

2. BACKGROUND: PROBABILISTIC DEMAND FORECASTING

The demand forecasting problem constitutes in predicting the demand for a group of items at a certain range of days in the future, given demand data for all items up to the present, as well as other input data sources. In a retail context, *demand* in the past typically refers to customer orders. Note that this is an approximation as demand is actually partially unobserved: orders for an item are subject to the item's availability. We represent demand at daily grain: $z_{it} \in \mathbb{N}$ for item i at day t. Users require probabilistic forecasts of total demand for a group of items $i \in \mathcal{I}$ at certain *lead times*[1] t_L and *spans* δ_S in the form of the *probability distribution* of

$$Z_{\mathcal{I};(t_L,\delta_S)} = \sum_{i \in \mathcal{I}} \sum_{t=t_L}^{t_L+\delta_S-1} z_{it} \quad (1)$$

A sensible requirement of a forecasting system is *consistency*: the predicted distribution of $Z_{\mathcal{I};(t_L,\delta_1+\delta_2)}$ should be the same as that of $Z_{\mathcal{I};(t_L,\delta_1)} + Z_{\mathcal{I};(t_L+\delta_1,\delta_2)}$. Moreover, users should be able to query any quantile of the forecast distribution, (e.g., the median or the 90-th percentile) for any combination of lead time and span, as well as any group of items $\mathcal{I}$. Forecasting draws from two different bodies of scientific work: (1) Time series methodology (ARIMA, exponential smoothing), developed in statistics and econometrics. These approaches are well researched, address extrapolation (time gap between training and test), temporal dynamics, and predictive distributions. Most methods are data-poor (so additional input data does not help much), and have few parameters to be learned. (2) Machine learning methods (classification, curve fitting): conceptually simple models are fit to training data, then evaluated on test data. Such techniques are data-rich, can ingest many data sources (e.g., price, holidays, brand, color). Training algorithms are

[1]Here, we assume that lead times t_L are absolute dates in the future, while spans δ_S are ranges (number of days).

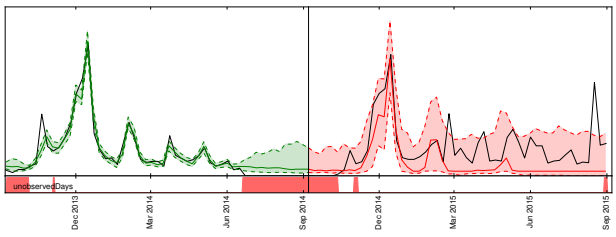

Figure 1: Visualization of a probabilistic demand forecast. The black line denotes the actual demand, while the green and red lines denote quantiles of the forecasted demand distribution (10th percentile, median and 90th percentile). Green lines denote the model samples in the training range, while the red lines show the actual probabilistic forecast on data unseen by the model. Note that demand can be partially unobserved (due to out-of-stock situations).

able to rapidly adapt to fluctuations in signals. However, extrapolation is not typically served, since training and test data are assumed to be i.i.d., and point predictions ('best guesses') are the norm.

Neither of the two approaches suffices in isolation for state-of-the-art forecasting in large catalogues. The key modelling challenge for us is to combine best practices from both disciplines, in order to provide probabilistic forecasts in a data-rich scenario. Further challenges include: (i) *Sales data is intermittent, often slow.* In very large catalogues, sets of items can exist that only sell very rarely, implying $z_{it} = 0$ for a majority of days. Yet, many of these can have intermittent sales: bursts separated by stretches of zeros. Standard time series methodology is ill-equipped for such data due to its inherent distributional assumptions. (ii) *Large number of new items.* At any given time, a large body of items can be new, being online for less than a season. In ML terms, we face a 'cold start' problem. It may therefore be required to learn linkages between new and established items. (iii) *Scale.* Large catalogues can contain millions of items, potentially in different marketplaces. For a typical demand history of hundreds of days and several dozens of features, a forecasting system therefore may face hundreds of billions of training points.

Let us highlight two modelling decisions we made. First, we deal with demand that has a non-standard distribution by introducing the so-called *multi-stage likelihood.* We use this likelihood in all our feature-based models, in particular generalized linear models (GLMs). At stage 0, a logistic regression model decides $z_{it} = 0$ versus $z_{it} > 0$, either emitting $z_{it} = 0$ or handing over to stage 1, where a second classifier decides $z_{it} = 1$ versus $z_{it} > 1$. Finally, if $z_{it} > 1$, we use Poisson regression on the transformed target $z_{it} - 2$. For GLMs with such a likelihood, we draw on well-researched, stable, numerically safe convex optimization approaches. Second, we represent distributional forecasts by *sample paths.* For each item i in a dataset, we sample demand values $[z_{it}]$ over a prediction range, and repeat this process many times. Distributional queries (such as a specific quantile for demand at a lead time of x weeks and a span of y weeks) are answered by averaging over the samples. Sample-based representations are flexible and easy to use, and queries are answered consistently. The potentially high storage costs are mitigated by exploiting the high sparsity of samples. Furthermore, sample paths also allow for easy combination of model outputs. As an example, Figure 1 shows a visualization of such a probabilistic forecast computed by our system.

3. SYSTEM ARCHITECTURE

In this section, we give a summary of the system architecture as illustrated in Figure 2. Our platform is implemented on top of Apache Spark [30] and leverages its popular abstraction for distributed computing. As already stated, we designed our platform as a unified system with a single codebase for experimentational and large-scale use cases. In the following, we describe its main components. All of these components are loosely coupled, so that they can run independently from each other, while exchanging data via a distributed filesystem.

Data Integration Component. The Data Integration Component allows for access to several external data sources, typically from distributed storage, provides data cleaning and enrichment, and joins the input data to a distributed de-normalized table where each row contains all data for an item.

Forecasting Component. The forecasting component is the "heart" of our platform. It consists of a routing component, a feature transformation component, learning algorithms, and an orchestration layer that leverages a highlevel dataflow abstraction to model ML pipelines. The routing component assigns groups of items to one or more dedicated learning algorithms. Each learner has a feature transformation flow as well as pre- and postprocessing logic associated with it. The feature transformation turns data into sparse matrices and provides customizable as well as standard transformations (crossing, binning, normalization, etc.) along the way. A learner invocation consists of a training and a prediction phase, where the former phase uses sparse linear algebra tools and convex optimization libraries, and the latter applies sampling methods. The final output of each learner run are sample paths as described in Section 2. The outputs of all learners are then consolidated into a single set of sample paths by means of ensembling. The forecasting component additionally supports generating reports which provide visualizations and summaries of learner internals.

Evaluation Component. The evaluation component consolidates all evaluation-related code in a central place. This is crucial to guarantee consistency and safe-guard against errors. For ML applications, errors in evaluations are much more grave than errors in models or algorithms, which is why extra care needs to be taken. Consistency, for example, is important in handling edge cases, and for non-standard definitions such as quantiles of count distributions. The evaluation component operates on sample paths generated by the forecast component. Sample paths allow for easy aggregation across time and items, and make quantile computation easy. Note however, that computing evaluations can be very expensive, therefore we typically compute a host of metrics at the same time to avoid multiple scans over the data, and persist the resulting evaluation data. Additionally, we treat reporting as a separate step that provides summarizations and visualizations of the evaluation data.

Output Generation Component. The output generation component consumes the sample paths produced by the forecasting component, enriches them with useful end-user information and allows us to convert the sample paths to several external formats.

Analysis/Research Layer. The analysis and research layer contains tools for real-time interaction and experimentation. An 'Interactive-Shell' powers interactive experiments as well as data exploration. It consists of an enriched Spark-shell with useful abstractions for common tasks such as data loading. Moreover, it also allows access to a custom, light-weight plotting library.

4. SYSTEM INTERNALS

Next, we turn our focus onto the details of the platform and discuss three features, which live on different levels on the system and in combination, allow us to form an end-to-end machine learning application: (i) A set of general distributed learning schemes, which allow us to integrate and scale-out a large class of learning algorithms (Section 4.1). (ii) Learners typically target specific bands of

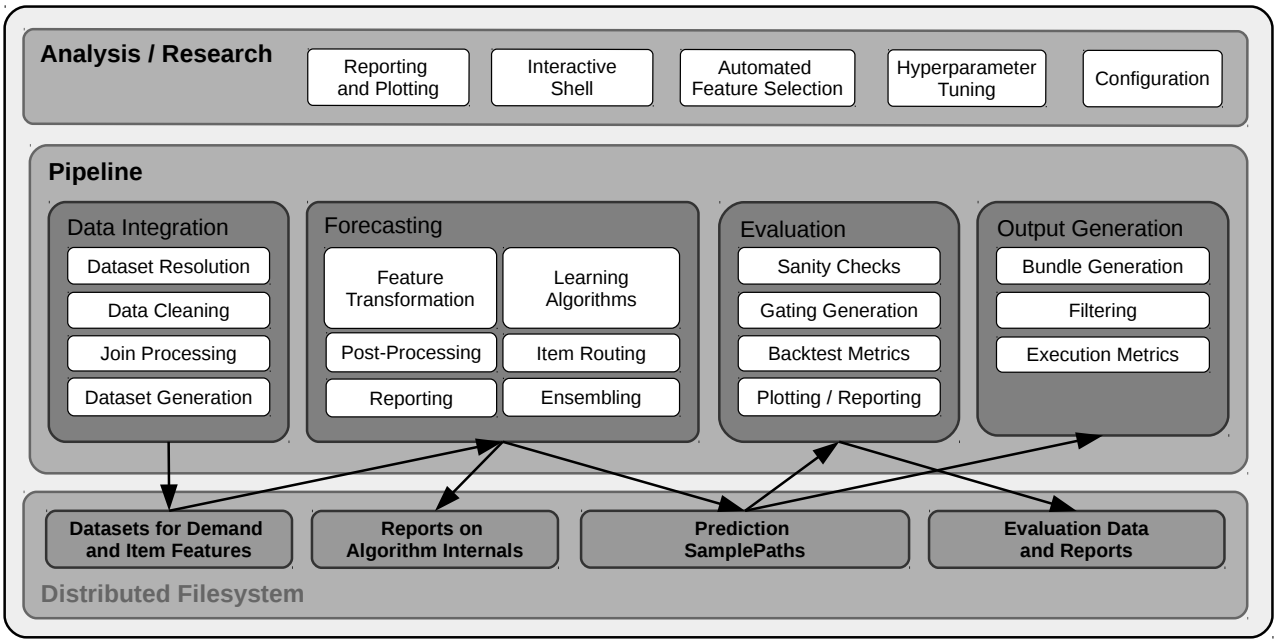

Figure 2: System Architecture: At its core, the platform consists of four loosely coupled components which all run on Apache Spark and exchange data via a distributed filesystem. The *Data Integration Component* fetches data from several data sources, cleans and joins this data and produces datasets comprised of item features and historical demand. The *Forecasting Component* contains most of the machine learning logic. It transforms the demand data into feature matrix representation and allows us to train a variety of forecasting models on the data. Finally, it ensembles the resulting predictions and generates a dataset of sample paths. The sample paths are consumed by the *Evaluation Component* and the *Output Generation Component*. The former allows us to run sanity checks on the predictions, to compute a huge variety of evaluation metrics and derive configurations for ensembles. The latter prepares the sample path data for potential external consumers. Furthermore, there is an orthogonal analysis component that allows users to run the forecasters in interactive mode and enables several model selection techniques such as automated feature selection.

items and new learners must be ramped-up slowly; therefore we provide flexible machinery to route items to learning algorithms (Section 4.2). (iii) Our system's different operation modes require us to define and execute complex end-to-end ML pipelines; we present a high-level dataflow abstraction to declaratively model such dataflows (Section 4.3).

4.1 Distributed Learning Schemes

In standard ML problems such as classification, the data typically consists of a large set of labeled observations, and we aim to learn a generalizing function to predict this label for unseen observations. In forecasting however, the ML problem at hand is more complex. We have to compute a probabilistic forecast for each individual item in the dataset, and we are provided with daily observations (in terms of demand or item features for the respective day) on a per-item level. As a result, we effectively have to handle time as additional dimension in the data compared to standard ML problems. Therefore, there exists a variety of approaches to tackle the forecasting problem: in classical approaches, a single (local) model per item time series is computed. However, in many cases it might be beneficial to employ global learners that consider groups of items, for example in order to tackle cold-start problems. Therefore, we employ a range of learning approaches that allow for different blendings of local and global learning. Our platform offers a variety of schemes for distributed learning of forecasting models, which learners leverage in every operation mode. Examples include conventional large-scale ML approaches like learning an individual model per instance in an embarassingly parallel manner, as well as learning a global model for all instances via batch gradient descent [9, 3]. Furthermore, we support two additional schemes which address the heterogenity of the data (e.g., different item groups). For the schemes discussed, we assume that the optimization problems we want to solve for learning forecasting models on a collection of items $\mathcal{I}$ compose of the following, well-known building blocks: a convex and differentiable function $l_i(w)$ for computing the loss with respect to the features X_i of a single item i and the model parameters w, as well as a convex regularizer $r(w)$. Regularization depends on a hyperparameter which we omit from our notation for the sake of simplicity. In case of the multi-stage likelihood and maximum likelihood learning, a typical choice is a squared loss function and an ℓ_2 regularizer.

Local learning. In this setting, we train a single model per item i parameterized by a weight vector w_i, independently of all other items. This means that we solve an optimization problem

$$\operatorname*{argmin}_{w_i} l_i(w_i) + r(w_i)$$

for every item i in isolation. We execute this scheme in a

simple data parallel manner with a *map* operator (Figure 3). In the map operator, the learner is provided with the features X_i related to an item i and learns the weight vector w_i for it.

While the parallel execution of this scheme is trivial, it allows us to easily scale out powerful algorithms not covered in this writeup, such as maximum likelihood parameter learning in state space models with non-Gaussian likelihood, using approximate Bayesian inference [26].

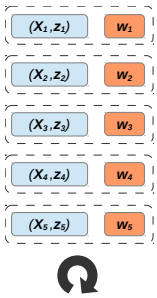

Figure 3: Local learning: embarassingly parallel execution with a *map* operator.

Global learning. Here we train a global model, parameterized by a weight vector w, for all items, resulting in the optimization problem

$$\underset{w}{\text{argmin}} \sum_{i \in \mathcal{I}} l_i(w) + r(w)$$

Our system applies distributed L-BFGS [7] to minimize this equation, an approach common in industry (e.g., the Mllib [20] library analogously trains generalized linear models). L-BFGS is executed via an iterative *map-reduce-update* scheme (Figure 4). At the beginning of an iteration, the current version of the weight vector w is broadcasted to all worker machines in the cluster. Next, each worker k computes the sum of the gradients $\sum_{i \in \mathcal{I}_k} \nabla l_i(w)$ for every item i in the worker's partition $\mathcal{I}_k$ of the data. This happens in parallel for all K partitions of the data (with $\bigcup_{k=1}^{K} \mathcal{I}_k = \mathcal{I}$ and $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset \ \ \forall i \neq j$). The parallel gradient computation is conducted via a *map* operation over the input. Next, the system runs a global aggregation via a *reduce* operator, in which the gradient contributions are summed up and combined with $\nabla r(w)$ on the master. The master then computes the next version of the weight vector w (using its approximation to the inverse Hessian and line search) and starts the subsequent iteration.

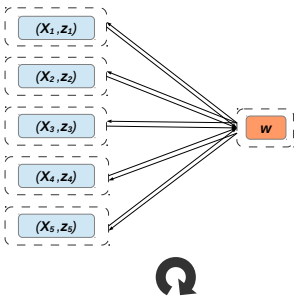

Figure 4: Global learning: distributed L-BFGS executed with iterative *map-reduce-update*

Local learning with hierarchical regularization. We support an extension to local learning which allows models to share statistical strength between items. The idea is to use a regularizer that shrinks the weight vector towards vectors learned at higher levels, incorporating the weights of many items. In the example, we employ a two-level hierarchy[2], introducing a set of parameter vectors w_g for $\mathcal{G}$ groups of items, as well as a global vector w. This hierarchy results in the following joint optimization problem for all items:

$$\underset{w,\{w_g\},\{w_i\}}{\text{argmin}} \Big[\sum_{i \in \mathcal{I}} l_i(w_i) + r(w_i - w_{g(i)} - w) \Big] + \sum_{g \in \mathcal{G}} r(w_g) + r(w)$$

We again employ an iterative *map-reduce-update* scheme for the learning procedure. We alternate between a local learning step for the item specific weight vectors w_i and a global hierarchical aggregation step to re-compute the group specific weights w_g and the global weight vector w (Figure 5). The current versions of the vectors w_g and w are broadcasted to the workers in the cluster, which execute the local learning step within a map operation. Our system conducts the subsequent hierarchical aggregation with a *reduce* operation that first re-computes the group specific vectors w_g from the item specific weights w_i and then re-computes the overall weights w from the group vectors w_g. For ℓ_2 regularization, these re-computations can be done in closed form. Finally, the updated set of vectors w_g and w are broadcasted to the workers to start the next iteration.

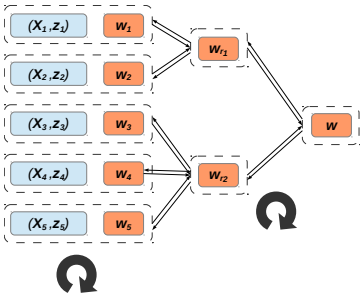

Figure 5: Local learning with hierarchical regularization: alternation between embarassingly parallel local learning and a global hierarchical aggregation step.

Global-local learning. The fourth distributed learning scheme comprises of a hybrid between global and local learning. The features as well as the weight vector which parameterizes the model here consist of a global part and a local part. In the case of the parameter vectors, the global part w is shared by all items and local parts v_i are specific to individual items i. Analogously, a global regularizer $R(w)$ and local regularizers $r_i(v_i)$ are applied, which leads to the optimization problem:

$$\underset{w,\{v_i\}}{\text{argmin}} R(w) + \Big[\sum_{i \in \mathcal{I}} l_i(w, v_i) + r_i(v_i) \Big]$$

On the technical side, this allows us to operate on a smaller shared vector than in the standard global setting. On the algorithmic side, this approach provides the freedom to use different sets of features for different kinds of items and to keep these features local to the items that exhibit them.

[2]note that the proposed approach allows for hierarchies of arbitrary depth

We employ a nested minimization approach, which we again execute via iterative *map-reduce-update* (Figure 6). In this nested minimization approach, we push the minimization over v_i into the sum, i.e.:

$$\underset{w}{\text{argmin}}\, R(w)+\sum_{i\in\mathcal{I}} \tilde{L}_i(w) \text{ with } \tilde{L}_i(w) := \min_{v_i} \Big[l_i(w,v_i)+r_i(v_i)\Big]$$

Note that the resulting global optimization problem is still convex, as minimization preserves convexity. In order to make this approach effective in practice, the inner optimization problems need to be solved as quickly as possible, as they are required for every evaluation of the global objective function (e.g., also during line searches in L-BFGS). In our experience it is sufficient to solve these inner problems to a fairly low tolerance and apply a hard limit on the number of inner iterations.

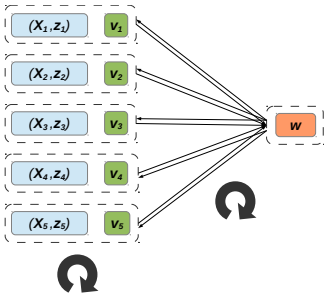

Figure 6: Global-local learning: Nested minimization, where every outer iteration re-computes shared global parts of the model and internally runs several embarassingly parallel inner iterations of learning the local parts of the model.

While individual learning algorithms form the core of our platform and dictate its prediction quality, they only constitute a single building block in a large end-to-end ML application from a systems perspective. In the following sections, we move up in the hierarchy and discuss our machinery for assigning items to learners and for modeling complex end-to-end ML pipelines.

4.2 Routing and Ensembling

Our platform has to be able to deal with numerous models which reflect the heterogeneity of the demand in many scenarios and the resulting diversity of the forecasting problem. For example, some models might be tailored towards specific groups of items such as new items, or items that share characteristics like seasonality. Furthermore, some models might only perform well during specific time periods in the prediction range, e.g., christmas. Usually, users try out a small set of baseline algorithms and add more specialized learners over time to improve prediction accuracy for certain subsets of items. Therefore, our system contains a routing component that controls the 'gating', the assignment of items to different learners both for training and prediction.

Gating. We leverage declarative rules comprised of predicates on item-based attributes (e.g., denoting the product group of an item) to specify the routing. These rules can either be manually defined or learnt from experimental evaluations on backtest datasets. The routing mechanism requires the definition of training as well as prediction routing rules to define the sets of items that a learner uses for training as well as for prediction. Here, the training set is often larger than the prediction set, especially for learners that embody information sharing between items (Section 4.1). The definition of the routing rules is a manual process, because the attributes used in the predicates of the routing rules must be determined before we derive the routing. Typical examples of attributes for routing predicates are the mean weekly demand or age of an item. More advanced routing attributes can be obtained from a previously trained classifier for example.

```
itemRouting {

  variableDefinitions { ... }

  training {
    GLMFast = [
      { satisfies = [velocity_medium],
        include = true },
      { satisfies = [velocity_fast],
        include = true },
      { include = false }],
    Base = [{ include = true }],
    Zero = [{ include = false }]
  }

  prediction {
    { satisfies = [likely_inactive],
      learners = { Zero = 1.0 } },
    ...
    { satisfies = [velocity_fast,
                   cat_PHONE, xmas],
      learners = { GLMFast = 0.5,
                   Base = 0.5 } }
    ...
    { learners = { Base = 1.0 } }
  }
}
```

Listing 1: Declarative routing configuration based on item attributes for an ensemble of three learning algorithms.

Generation of Gating Rules. Given the routing attributes, we employ an automated way to derive the gating for a set of pre-determined buckets (subsets of items). For each such bucket, we generate a forecast from every learner for all contained items, compute aggregate evaluation metrics (e.g., the aggregate P90 quantile loss), and finally pick the best performing learner.

Ensembling. This gating mechanism also enables users to leverage multiple learners for predicting the demand of a single item. The main usage scenario of this functionality is to generate ensemble predictions from different learners. Since learners emit samples as prediction output in our architecture, ensembling boils down to mixing the samples obtained from different learners. Another important function of the ensembling mechanism is to provide fallback predictions in rare cases where an individual learner fails to generate samples for an item.

Example. Listing 1 illustrates a small hypothetical example of a declarative routing configuration. The configuration starts with the definition of boolean variables which will be used to decide upon the routing of items, contained in the `variableDefinitions` block. We omit its details for brevity, and describe the variables used in the example configuration. First, there is a set of variables called `velocity_slow`, `velocity_medium` and `velocity_fast` denoting in which pre-

defined range of mean weekly historical sales an item falls. Next, there are catalogue specific variables, e.g., `cat_PHONE` which denotes that an item belongs to the phone category. Furthermore, there are complex variables such as `likely_inactive` whose value depends on the output of a previously trained classifier that predicts if an item will not sell anymore (e.g., because the item is not produced anymore, but the system is not aware of this information). Additionally, there is the attribute `xmas` which does not depend on the item itself, but on specific dates in the forecasting range, e.g. a couple of weeks before christmas.

We employ three learning algorithms in the example: an algorithm called `Zero`, which constantly predicts zero sales (and is only useful for inactive items), a simple baseline learner called `Base`, and a GLM with hierarchical regularization (Section 4.1) denoted `GLMFast`, which we assume is tuned for high velocity items. The `training` block dictates the assignment of items to learning algorithms in the training phase. A list of statements must be provided for every configured learner. Then, for each item in the dataset, the statements are processed as follows: we find the first statement for which the conjunction of variables in the `satisfies` clause evaluates to true, and include the item in the training set for the learning depending on the value of the corresponding `include` clause. In the example, the training set for the GLM consists of all items with fast or medium velocity, the baseline learner is trained on all items, and the zero forecaster on none, as it does not require training.

The `prediction` block consists of a list of statements, which are evaluated for every item and every day in the forecasting range. In contrast to the training block, an item can only have a single match here. The first statement with a matching `satisfies` clause defines how the predictions samples for the item on the respective day are generated: we compute prediction samples from all the learning algorithms specified in the `learners` clause and combine the samples according to the given probability distribution. In our example, we gate all the items that received the `likely_inactive` symbol to the `Zero` model so that we get a prediction of constant zeros. The next clause dictates that during the christmas period all phone items (`cat_PHONE`) with a high number of sales in the past (as indicated by the `velocity_fast` symbol) get a 50/50 mix of predictions from the baseline model and the GLM. Finally all items that have not been captured by any statement yet receive their predictions from the baseline model (as indicated by the last statement). A further noteworthy advantage of this declarative routing machinery is that we can compute a complete table of item assignments which is very useful for debugging and reporting.

4.3 High-Level Dataflow Abstraction

While training models and assigning sets of items to them is crucial for a demand forecasting system, actual workloads comprise of many more operations (e.g., pre-processing, report generation, persistence of intermediate results, etc.), that need to be orchestrated, typically in the form of some kind of pipeline. It has been acknowleged that implementing complex ML pipelines for real-world systems poses a huge challenge [25]. Static pipelines usually lack flexibility: although their behavior can be partially influenced through configuration values, adapting these pipelines to certain operation modes such as ensembles of many learning algorithms becomes difficult. Over time, pipelines have a tendency to become increasingly complex: sources of complexity in our scenario are the need for backtests on many different forecast start dates or the comparison of many ensemble configurations. Furthermore, static pipelines sometimes impose performance overheads, as they apply hardcoded execution strategies, typically aimed at guaranteeing robust runs on the whole input data. However, in ad-hoc execution modes on small datasets, these strategies can often cause redundant work (e.g., the re-computation of predictions or the repetition of feature transformations for many learners which operate on the same features). Moreover, certain system functionality, such as the routing, is not required in ad-hoc operation modes such as experiments with single learners. While some researchers propose to define custom domain-specific languages for such scenarios [16], we choose a more lightweight path by implementing an additional dataflow abstraction on top of Spark, which resides on a lower level than a custom language, but still enables flexible modeling and a set of optimizations. This dataflow abstraction enables us to lazily declare a pipeline, which the system executes afterwards. During execution, the system can automatically apply inspections and optimizations. As a side effect, the resulting abstraction enforces sound engineering principles such as encapsulation and separation of concerns.

Dataflow Abstraction. Dataflows have proven themselves as a useful abstraction for general distributed, data-intensive computing [8, 1, 6]. We therefore introduce a simple dataflow abstraction for our forecasting flows to model our platform's different operation modes. We define a *flow* as a directed, acyclic graph (DAG), denoted $G = (V, E)$, where the vertex set V is comprised of so-called *sources* and *operations*, and the edges E denote flow of data between vertices. A *source* represents a second-order function $f_r \rightarrow O$, where the user-defined first-order function (UDF) f_r produces an output of type O. An *operation* is a second-order function $I \times f_o \rightarrow O$, which given an input of type I and a user-defined first-order function $f_o : I \rightarrow O$ produces an output of type O. All supplied UDFs must be deterministic and side-effect free. Furthermore, we provide a set of *materialization operations*, which are functions: $I \rightarrow I$ with pre-defined side effects (such as persisting a dataset in the DFS or caching it in memory). Operations can contain distributed as well as non-distributed computations. Naturally, the output type of the operations pointing to a vertex must match its input type, (or the combination of the output types, in case of multiple incoming edges). Flows are lazily constructed and executed by the system afterwards. The runtime executes a flow by recursively invoking the operations, similar to a depth-first walk starting from the leaf vertices of the DAG. The main components of our dataflow implementation are the abstract types `Source` and `Operation`. These are extended to implement different operations common to the ML pipeline. Flows are built from a composition of sources and operations, which applies type checks at compile time. The composability of the flow allows for a simple way to create different versions of our pipeline. Routing is not needed in single learner experiments for example, and we can define a flow that simply omits this step.

Automated Inspection & Optimization. Dataflow abstractions are commonly used to apply automatic optimizations to programs and queries (e.g, query optimization

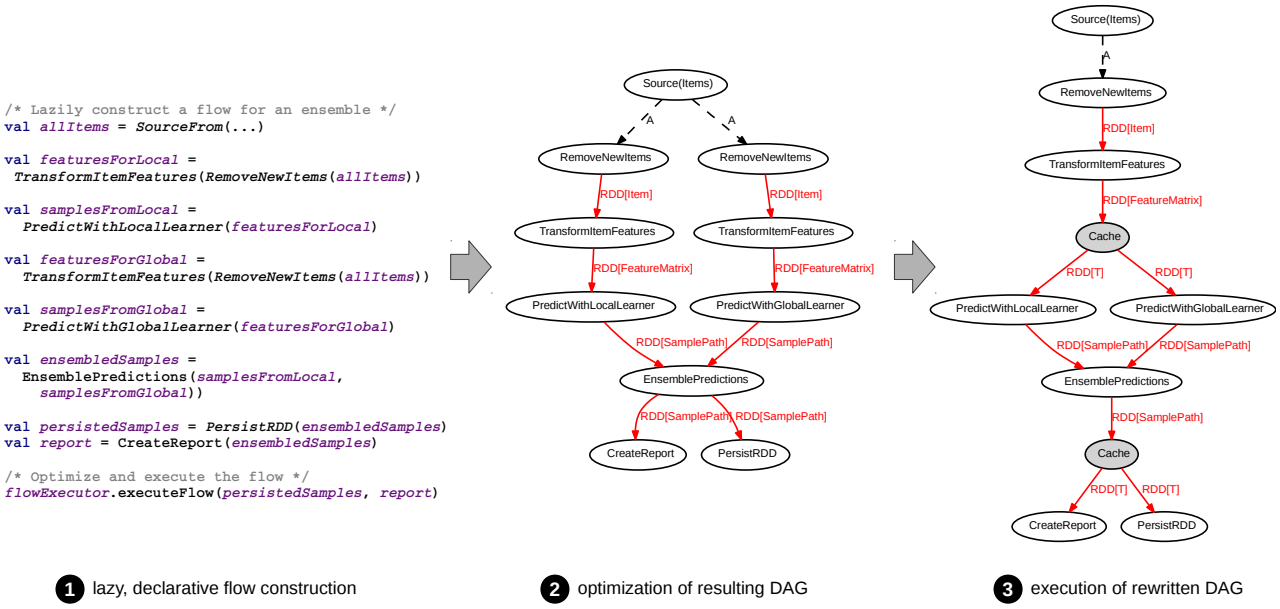

Figure 7: Toy example for ML dataflows in our platform: (1) ML dataflows are defined lazily in code by composing sources and operators. Finally, the sinks of the constructed DAG are passed to an executor. (2) Our system operates on the intermediate DAG representation and applies optimization techniques such as common subexpression elimination and cache point selection. In this example, the duplicated operators `RemoveNewItems` and `TransformItemFeatures` are merged into a single pipeline, and a cache operator is injected after all operators whose `RDD` output is read more than once. (3) The rewritten DAG is executed on Apache Spark.

in relational databases), usually by having the system reason about the computation using algebraic properties of the operators. While our proposed abstraction lacks algebraic properties, it still enables a set of general inspections and optimizations (similar to the functionality offered in KeystoneML [27]). For example, one of the biggest performance hazards in Spark-based applications is the implicitly triggered re-computation of distributed datasets that occurs if a non-cached distributed dataset is accessed multiple times (which even worse, can also trigger transitive re-computations). In a complex application, where references to distributed datasets are passed around frequently between different modules, its often not possible to locally (e.g., inside a method body) make all required caching decisions, as putting one dataset into the cache can result in the unwanted eviction of other data; furthermore potential performance gains also depend on the shape of the input data and the cluster. We are not aware of an easy-to-use automatic way that Spark provides to even detect these re-computations other than manually parsing them from its event log. Our operator abstraction on the other hand allows us to detect re-computations in cases, where an operator outputs a distributed dataset. Our runtime, which executes the dataflow, can detect such outputs during the depth-first walk through the DAG, check whether the dataset is currently cached (by interacting with the Spark cache), and record the number of consumptions of uncached operation outputs. A current limitation of this approach is that it cannot detect re-computations inside UDFs.

Furthermore, backtests and ensemble computations, which involve multiple learners, require us to execute tasks multiple times, such as reading the input data, filtering it according to routing rules and transforming the data to matrices. In many cases, this work is redundant, e.g., when we train different learners which use the same feature set. The dataflow abstraction allows us to detect such cases. By definition, the output of an operation is determined only by its UDF and its inputs. We can identify common operator subtrees (that have sources as leafs) in the DAGs for multiple learners, execute the subtree only once and re-use the result afterwards. Note that this is a special case of the well known common subexpression elimination applied by compilers. Another optimization targets a long-term goal of the dataflow abstraction: we want to have the system inject performance-critical materialization operators (e.g., for caching) automatically in suitable places in the dataflow. This allows the system to tackle the re-computation problem. Such an injection is not trivial however, as adding cache points is not always beneficial, e.g. there might be cases where it is cheaper to re-compute data and the memory consumed by the caching is also not available for other expensive system operations such as distributed shuffles. We experiment with a simple approach for cache point injection, where we inject a cache operator after every operator whose output is consumed more than once. However, we find that this does not reliably decrease runtime, which we attribute to the fact that Spark's cache competes for memory within its 'block manager' that also holds the results of distributed shuffles.

Example. Figure 7 shows a toy example of computing ensemble predictions from two learners based on our dataflow abstraction. The Scala code on the left side illustrates how a flow is lazily constructed from a set of predefined operators which internally execute several Spark operations. In

the example, a set of items is read via the `SourceFrom` operation. Next, the pipeline for a fictitious 'global learner' is set up: we remove new items with the `RemoveNewItems` operator, apply feature transformations with a `TransformItemFeatures` operator, and finally compute predictions by invoking the `PredictWithGlobalLearner` operator. We define a similar pipeline for a second fictitious 'local learner'. The output of these learners is consumed by an `EnsemblePredictions` operator, which combines them according to some ensemble definition. Finally, the ensembled predictions are written to stable storage by the `PersistRDD` operator, and a report is generated for them by a `CreateReport` operator. In the end, we instruct the system to execute the lazily constructed dataflow via the `flowExecutor.execute(...)` instruction, which is presented with the sinks of the dataflow graph. Internally, our system converts the code to a DAG representation and applies the aforementioned optimizations. The subgraphs for both learners share the operator subexpression `SourceFrom` → `RemoveNewItems` → `TransformItemFeatures`, which gets merged into a single instance whose output is consumed by both learners. Next, the system applies a simple cache point injection strategy: we inject a cache operator after any operator that produces an RDD which is read more than once (and therefore might be re-computed). In our example such operators are `TransformItemFeatures` and `EnsemblePredictions` The resulting rewritten DAG, which the system will execute, is shown on the right of Figure 7.

5. EVALUATION

For our experimental evaluation, we focus on the distributed learning schemes from Section 4.1. All experiments are run using Spark 1.4, Hadoop 2.4 and the reference L-BFGS-B Fortran implementation. To the best of our knowledge, there exists no publicly available forecasting system that would be suitable for comparing to ours.

5.1 Scalability

We investigate the scalability of the scheme for local learning with hierarchical regularization, described in Section 4.1. We decide to concentrate on this scheme because we find it to be the most popular choice amongst our users; furthermore, the scalability of the global learning scheme is already well researched [4]. In our instantiation, the model applies a two-level hierarchy, where the first level is based on item categories and the second level comprises a global regularization. Locally, we learn generalized linear models for the multi-stage likelihood described in Section 2, using L-BFGS [7] as optimization algorithm and three outer iterations per stage.

All scalability experiments are run using 'm3.2xlarge' EC2 instances with 8 cores and 30 GB RAM each, and data is read from the S3 filesystem. The input data to the system for each item is its demand history plus a matrix of ~ 300 sparse features for a year in the past. We train three different models for each item (which together allow us to sample from our multi-stage distribution). We configure Spark to run with a degree of parallelism equal to the number of worker cores in the cluster, as our workload is dominated by computationally intensive learning operations.

Scalability with increasing data size. We fix the cluster size to 25 machines, execute the model computation with the hierarchical learner for an increasing number of items (from one to five million), and measure the runtime of the model training phase, whose input are item feature matrices, which are materialized in memory. Figure 8 shows the resulting execution times, ranging from 232 seconds to 832 seconds. We observe a linear running time increase, which is what we expect, as the majority of work is embarassingly parallel local model training, and distributed computations are only necessary for the aggregation steps that re-compute the hierarchical regularization vectors. These distributed aggregations are fast, because we execute them with a tree aggregation on the relatively small weight vectors.

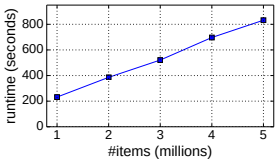

Figure 8: Linear speedup for an increasing number of input items on a fixed-size cluster of 25 machines.

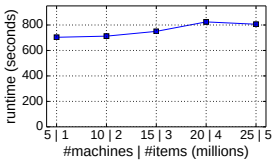

Figure 9: Effect of proportionally increasing cluster and data size.

Scalability with increasing data and cluster size. Next, we focus on the scalability effects of increasing the number of worker machines in the cluster proportionally to a growing amount of input items. We start with five workers and one million items and increase this workload up to 25 worker machines and five million items. Figure 9 illustrates the resulting mean execution times. Ideally, we would see a constant runtime. However, its is not possible to reach this ideal scale-out for many reasons (such as network overheads). This effect has also been observed in other large-scale ML systems [11, 24]. Nevertheless, we see that our system achieves a steady controlled increase in execution time with growing data and cluster size.

Impact of increasing the time dimension of the feature matrices. In this last scalability experiment, we investigate the effects of increasing the time dimension of the feature matrices for the items in our forecasting workloads. We use a fixed cluster size of 25 machines, a fixed set of four million items and vary the size of the feature matrices we generate. In order to achieve this variation, we have our system generate feature matrices for different history lengths of 90, 180, 270 and finally 365 days. Figure 10 shows the resulting runtimes. We again see a linear running time increase, yet increasing the time dimension of the feature matrices has a much weaker impact on the runtime than increasing the number of items. This is because the time dimension of the feature matrices only affects the the local learning steps. The dimensionality of the weight vectors is not impacted by that, therefore there is no reduction of the amount of distributed work that has to be conducted. This means that our system is well suited to handle increasing item histories.

5.2 Adaptability to Optimization Algorithms

We showcase the power of the abstraction underlying our distributed learning schemes, which supports users in separating the modelling of the ML problem from the actual learning procedure. Here we choose the global-local optimization problem introduced in Section 4.1 as example. In our codebase, users can implement predefined interfaces

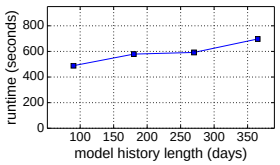

Figure 10: Modest linear runtime increase for growing item histories.

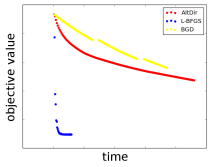

Figure 11: Convergence rates of alternating directions, batch gradient descent and nested minimization (axis labels omitted due to non-public nature of the target metric).

for modelling the ML problem at hand (e.g., their implementation has to compute value and derivatives of the loss function for individual items), and can choose from an existing set of optimization algorithms (or implement their own) for learning the parameters of the ML problem. For the global-local problem, we compare the nested minimization approach detailed in Section 4.1 to standard batch gradient descent (BGD) and an 'alternating directions' (AltDir) approach. In the latter approach, we alternate between partially optimizing the global-local optimization problem with respect to the global parameters w (keeping all local parameters v_i fixed), and optimizing for all local parameters v_i while keeping w fixed. As the optimization problem decomposes over the local v_i, the latter step can be performed independently for each item i in an embarassingly parallel manner.

For the comparison, we set up the problem with ~ 1000 sparse global features and ~ 10 dense local features, and apply it to ~ 1 million items each with about 800 days of training data. We run this set of experiments on a cluster consisting of 100 4-core EC2 instances with 32 GB RAM each. Both the alternating and the nested minimization use 10 inner iterations for each outer iteration. The results are shown in Figure 11. We see that the nested approach using L-BFGS converges much more quickly than the other two algorithms. We attribute the fact that the nested approach outperforms the alternating approach so drastically to its ability to re-use the approximation to the inverse Hessian it has built up throughout the entire optimization.

6. LEARNINGS

We summarize several of our learnings from building this platform, both technical and operational in nature.

Shortcomings of Apache Spark. Our platform is built on Apache Spark, which in principle allows running large-scale data processing workloads on compute clusters of varying size with the same code. In practice, this can be difficult, as Spark programs represent hardcoded physical execution plans that are usually tailored to run robustly in production setups, and therefore naturally result in huge overheads when run on smaller data. Spark is built on the idea of repeatable transformations on immutable data structures [30], which greatly simplifies scheduling and fault tolerance. Yet, we encountered cases where a carefully tailored way to update state local to a machine instead of always producing copies would be very beneficial in terms of performance (e.g., in the repeated updates for the local state of the global-local learning scheme in Section 4.1).

Benefits of declarative programmable learner gating. We discuss our experiences in working with the declarative routing mechanism described in Section 4.2. A major challenge in most real-world ML applications is the highly heterogeneous nature of real-world data. We described some of the reasons for heterogeneity in demand data in Section 2. A common pattern that we observed in usage of our system is that people first establish cheap baseline methods that produce forecasts for all items in a dataset, and later concentrate on more specialized algorithms to improve the prediction accuracy for certain important subsets of the data. Such an approach is a perfect use case for our gating machinery, which allows users to specify such assignments in a declarative and easily maintainable way. Furthermore, the gating process can be automated by generating the gating rules from experimentation results as we describe in Section 4.2. If ensembling is used, the gating rules allow for a simple way to communicate the provenance of a computed prediction to consumers of these forecasts. Another major benefit of the routing mechanism for our users is performance optimization, since it allows users to run complex and computationally expensive learners only on small, important subsets of items, while the bulk of items is routed to cheaper baseline learners.

Automation of model selection. Another operational problem faced in almost any ML scenario is the automation of model selection, e.g., to parameterize ML models for different deployment scenarios. Our system has a high degree of automation via tools such as greedy forward feature selection and hyper-parameter optimization via Bayesian optimization approaches. Nevertheless, a fair bit of manual work is still needed. Further automatization hinges on a trusted objective function and an ability to iterate quickly over many models. While we have identified scores for some sub-problems, it is an open question to find a score that captures all human intuition on what makes a forecast good. Even with a perfect score, the scale of many problems faced in industry use cases incurs potentially long runtimes of expensive learners and thus further prevents the system from "killing the problem with iron", e.g., from simply running extensive grid searches for all incorporated models. Finding smart ways to reduce the runtime of model selection workloads is a general problem however, and currently becoming an active area of research [16].

7. RELATED WORK

We compare our system to previously proposed platforms for large-scale machine learning from the research world, and consider the question whether these systems would have been a suitable base to implement our use case on top of. Afterwards, we review related work on data management for end-to-end machine learning, and finally briefly discuss related literature on forecasting problems. A set of general platforms for distributed machine learning have been proposed in recent years. *SystemML* [11] allows users to specify ML programs in a declarative, R-like language. These programs will automatically be optimized and executed by a massively parallel dataflow system running on a cluster. In contrast to most scalable ML systems, SystemML internally uses blocked matrices as physical data representation, and

its operators are built to consume and produce such matrix blocks. Featurewise, SystemML could have been a candidate system for our implementation, yet it was not publicly available when the development on our platform began, and its language provides much less flexibility in comparison to Scala (especially for data integration tasks like ingestion from internal formats). Furthermore, there is no explicit support for time series data in SystemML. A related system is *Samsara* [22], which offers a Scala DSL for distributed matrix operations, optimizes the resulting programs, and runs them on a dataflow system. While Samsara allows for easier integration of external code than SystemML, due to it running natively in Scala, it also does not offer dedicated support for time series data.

MLBase [15] is a proposal for a comprehensive distributed ML platform built on *Apache Spark* [30]. The vision was to provide users with machinery for simple and declarative specification of ML tasks. Internally, these tasks should then be translated to so-called 'logical learning plans'. An optimizer conducts automatic model selection for these plans to determine a well working algorithm, hyperparameters and features for the ML task. The proposed vision resulted in the development of the *Mllib* library [20]. Mllib contains a collection of carefully implemented, scalable algorithms for standard machine learning problems such as classification, regression and recommendation mining. It also allows its users to declaratively define ML 'pipelines' consisting of steps like feature generation, learning or evaluation. In contrast to SystemML, it does not allow for the declarative specification of algorithms however. In our experience, experimentation, algorithm debugging and model selection are the most time consuming and difficult to scale tasks in a large-scale ML system, and in our case call for a far more advanced machinery than what Mllib has to offer (e.g., because different buying scenarios need to be taken into account in the evaluation of forecasts). Mllib's main goal is to provide a mature collection of standard algorithms, which conflicts with our use case that is very different from standard ML problems, and requires the design of custom algorithms. *Graphlab* [18] is a fast, distributed ML platform, built on the idea of asynchronously scheduling model updates based on the dependency graph of the data. However, the graph-parallel abstraction does not match our forecasting models well. Furthermore, Graphlab lacks general abstractions for distributed data processing to handle tasks like feature extraction and feature transformation.

Scientists are becoming more and more aware of challenging research questions of building and maintaining complex, real world, end-to-end ML systems [25, 17, 23]. While the initial research focus has been on the efficient training of ML models on large datasets, a set of orthogonal problems is currently being identified and addressed. Examples include the automated selection, management and provenance tracking of ML models [16, 29]. Another field of interest that we also faced in our system (see Section 4.3) is the modeling and execution of complex ML workflows. Many approaches [20, 27] build on variants of the popular pipeline abstraction in *scikit learn* [21]. Recent work addresses questions like the automatic optimization of such pipelines [27] and efficient ways to execute different versions of the contained operators [28]. *KeystoneML* [27] offers a larger set of optimizations than our dataflow abstraction, however it lacks support for routing and ensembling, which quickly become important when working with heterogeneous real-world datasets.

In contrast to general purpose machine learning platforms and systems, we are not aware of public literature on demand forecasting from a platforms/systems perspective apart. Literature is limited to technology overviews [12] or white papers on industry solutions by enterprise software/analytics providers such as BlueYonder, Oracle, SAP or SAS. These solutions do not address the scale and particularity of our demand forecasting problem. However, general interest in demand forecasting seems to be rising, e.g., [13, 14].

8. CONCLUSION

We have presented a novel retail demand forecasting system which contains state-of-the-art machine learning approaches and a rich infrastructure to support experimentation. At the same time, it meets the requirements of production use cases. We highlight three general learnings from our work: (i) It is greatly beneficial to have a single codebase for scalable execution and experimentation. Over the course of its history, this approach has proven to be very powerful by enabling rapid agile development. (ii) Following sound ML principles has allowed us to reach great accuracy in forecasting and extend our system to many different learners. (iii) A strong focus on experimentation and backtests is crucial, as these scenarios are most demanding in terms of resource consumption and scalability.

From our experience in developing the platform however, we also conclude that there is still a tremendous amount of systems knowledge necessary for building scalable machine learning systems. A single unfortunate choice of a join strategy or materialization point in a dataflow can have severe negative effects on the scalability and runtime of an algorithm implementation. This problem is deeply related to the fact that Apache Spark still requires its users to hard-code physical plans when programming via the resilient distributed datasets abtracion. Typically, the resulting execution plans are designed to be robust on large datasets in demanding scenarios, which means that they result in overheads when one runs such pipelines on small subsets of the input data for experimentation purposes. In many cases, it is difficult to switch such workloads to abstractions that allow Spark to optimize the computations (such as `DataFrames` [2]), as forecasting code needs to work with many complex objects, e.g., feature matrices representing different feature vectors and time-series, which are difficult to fit into a denormalized, relational representation.

In order to increase both the flexibility and performance of our system, we aim to build on recent research for optimizing ML workloads, especially for experimental tasks. Examples of such optimization aims include the increase of task parallelism [5] as well as enabling intelligent materialization and re-use of intermediate results [31, 16]. Additionally, we will explore how to integrate deep neural network learners that promise to be more accurate than state-of-the-art models, while requiring minimal feature engineering [10].

9. ACKNOWLEDGEMENTS

We would like to thank Ed Banti, Alessya Labzhinova, Telmo Menezes, Borys Marchenko, Thoralf Klein, Kokhi Nishio, Victor Suen, Syama Rangapuram, Simone Forte, Jean-Baptiste Cordonnier, Johannes Kirschnick and Stephan Seufert for their contributions.

10. REFERENCES

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[3] M. Bilenko, T. Finley, S. Katzenberger, S. Kochman, D. Mahajan, S. Narayanamurthy, J. Wang, S. Wang, and M. Weimer. Towards Production-Grade, Platform-Independent Distributed ML. In *Machine Learning Systems Workshop at ICML*, 2016.

[4] M. Bilenko, T. Finley, S. Katzenberger, S. Kochman, D. Mahajan, S. Narayanamurthy, J. Wang, S. Wang, and M. Weimer. Towards Production-Grade, Platform-Independent Distributed ML. In *Machine Learning Systems Workshop at ICML*, 2016.

[5] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7):553–564, 2014.

[6] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[7] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal*, 16(5):1190–1208, 1995.

[8] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[9] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *NIPS*, 19:281, 2007.

[10] V. Flunkert, D. Salinas, and J. Gasthaus. DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks. *arXiv preprint arXiv:1704.04110*, 2017.

[11] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[12] T. Januschowski, S. Kolassa, M. Lorenz, and C. Schwarz. Forecasting with in-memory technology. *Foresight*, 2013.

[13] A. Jha, S. Ray, B. Seaman, and I. S. Dhillon. Clustering to forecast sparse time-series data. In *ICDE*, 2015.

[14] kaggle.com. Rossmann store sales. https://www.kaggle.com/c/rossmann-store-sales.

[15] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[16] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 2015.

[17] J. Lin and A. Kolcz. Large-scale machine learning at twitter. In *SIGMOD*, pages 793–804, 2012.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB*, 5(8):716–727, 2012.

[19] S. Makridakis, S. C. Wheelwright, and R. J. Hyndman. *Forecasting methods and applications*. John Wiley & Sons, 2008.

[20] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, 2011.

[22] S. Schelter, A. Palumbo, S. Quinn, S. Marthi, and A. Musselman. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. In *Machine Learning Systems Workshop at NIPS*, 2016.

[23] S. Schelter, V. Satuluri, and R. Zadeh. Factorbird - a parameter server approach to distributed matrix factorization. *Distributed Machine Learning and Matrix Computations Workshop at NIPS*, 2014.

[24] S. Schelter, J. Soto, V. Markl, D. Burdick, B. Reinwald, and A. Evfimievski. Efficient sample generation for scalable meta learning. In *ICDE*, pages 1191–1202, 2015.

[25] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *NIPS*, pages 2503–2511, 2015.

[26] M. Seeger, D. Salinas, and V. Valentin Flunkert. Bayesian Intermittent Demand Forecasting for Large Inventories. In *NIPS*, 2016.

[27] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. *ICDE*, 2017.

[28] T. Van der Weide, O. Smirnov, M. Zielinski, D. Papadopoulos, and T. van Kasteren. Versioned machine learning pipelines for batch experimentation. In *Machine Learning Systems workshop at NIPS*, 2016.

[29] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: A system for machine learning model management. In *Workshop on Human-In-the-Loop Data Analytics at SIGMOD*, pages 14:1–14:3, 2016.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

[31] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276. ACM, 2014.