The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox

Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li,
Zhao Zhang, Michael J. Franklin, Ali Ghodsi, Michael I. Jordan
UC Berkeley AMPLab

ABSTRACT

To enable complex data-intensive applications such as personalized recommendations, targeted advertising, and intelligent services, the data management community has focused heavily on the design of systems to train complex models on large datasets. Unfortunately, the design of these systems largely ignores a critical component of the overall analytics process: the serving and management of models at scale. In this work, we present Velox, a new component of the Berkeley Data Analytics Stack. Velox is a data management system for facilitating the next steps in real-world, large-scale analytics pipelines: online model management, maintenance, and serving. Velox provides end-user applications and services with a low-latency, intuitive interface to models, transforming the raw statistical models currently trained using existing offline large-scale compute frameworks into full-blown, end-to-end data products capable of targeting advertisements, recommending products, and personalizing web content. To provide up-to-date results for these complex models, Velox also facilitates lightweight online model maintenance and selection (i.e., dynamic weighting). In this paper, we describe the challenges and architectural considerations required to achieve this functionality, including the abilities to span online and offline systems, to adaptively adjust model materialization strategies, and to exploit inherent statistical properties such as model error tolerance, all while operating at "Big Data" scale.

1. INTRODUCTION

The rise of large-scale commodity cluster compute frameworks has enabled the increased use of complex analytics tasks at unprecedented scale. A large subset of these complex tasks, which we call *model training* tasks, facilitate the production of statistical models that can be used to make predictions about the world, as in applications such as personalized recommendations, targeted advertising, and intelligent services. By providing scalable platforms for high-volume data analysis, systems such as Hadoop [2] and Spark [21] have created a valuable ecosystem of distributed model training processes that were previously confined to an analyst's R console or otherwise relegated to proprietary data-parallel warehousing engines. The database and broader systems community has expended considerable energy designing, implementing, and optimizing these frameworks, both in academia and industry.

This otherwise productive focus on model training has overlooked a critical component of real-world analytics pipelines: namely, how are trained models actually deployed, served, and managed? Consider the implementation of a collaborative filtering model to recommend songs for an online music service. We could use a data-parallel modeling interface, such as MLbase [11] on Spark [21], to build a model of user preferences (say, in the form of a matrix representing predictions for user-item pairs) based on historical data—a batch-oriented task for which Spark is well suited and optimized. However, if we wish to actually *use* this model to deliver predictions on demand (e.g., as part of a web service) on interactive timescales, a data-parallel compute framework such as Spark is the wrong solution. Batch-oriented designs sacrifice latency for throughput, while the mid-query fault tolerance guarantees provided by modern cluster compute frameworks are overkill and too costly for fine-grained jobs. Instead, the overwhelming trend in industry is to simply dump computed models into general-purpose data stores that have no knowledge of the model semantics. The role of interpreting and serving models is relegated to another set of application-level services, and the management of the machine learning life cycle (Figure 1) is performed by yet another separate control procedure tasked with model refresh and maintenance.

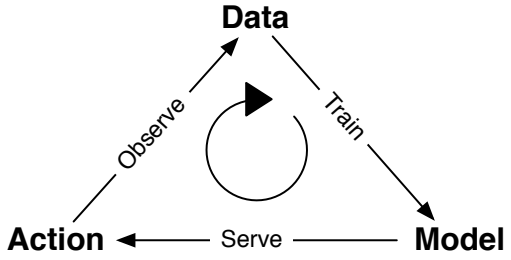

Figure 1: **Velox Machine Learning Lifecycle** Velox manages the ML model lifecycle, from model training on raw data (the focus of many existing "Big Data" analytics training systems) to the actual actions and predictions that the models inform. Actions produce additional observations that, in turn, lead to further model training. Velox facilitates this cycle by providing functionality to support the missing components in existing analytics stacks.

As a data management community, it is time to address this missing piece in complex analytics pipelines: machine learning model management and serving at scale. Towards this end, we present Velox, a model management platform within the Berkeley Data Analytics Stack (BDAS). In a sense, BDAS is prototypical of the real-world data pipelines above: prior to Velox, BDAS contained a data storage manager [13], a dataflow execution engine [21], a stream processor, a sampling engine, and various advanced analytics packages [17]. However, BDAS lacked any means of actually serving this data to end-users, and the many industrial users of the stack (e.g., Yahoo!, Baidu, Alibaba, Quantifind) rolled their own solutions to model serving and management. Velox fills this gap.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

Specifically, Velox provides end-user applications with a low-latency, intuitive interface to models at scale, transforming the raw statistical models computed in Spark into full-blown, end-to-end data products. Given a description of the statistical model expressed as a Spark UDF, Velox performs two key tasks. First, Velox exposes the model as a service through a generic model serving API providing low latency predictions for a range of important query types. Second, Velox keeps the models up-to-date by implementing a range of both offline and online incremental maintenance strategies that leverage both advances in large-scale cluster compute frameworks as well as online and bandit-based learning algorithms.

In the remainder of this paper we present the design of the Velox system and present observations from our initial, pre-alpha implementation. In Section 2, we describe the key challenges in the design of data products and outline how Velox addresses each. In Section 3, we provide an overview of the Velox system, including the serving, maintenance, and modeling components. We discuss our current design and implementation of each in Sections 4 through 6. Finally, we survey related work in Section 7 and conclude with a discussion of ongoing and future work in Section 8.

2. BACKGROUND AND MOTIVATION

Many of today's large scale complex analytics tasks are performed in service of *data products*: applications powered by a combination of machine learning and large amounts of input data. These data products are used in a diverse array of settings ranging from targeting ads and blocking fraudulent transactions to personalized search, real-time automated language translation, and digital assistants.

An example application: To illustrate the implications of data product design on data management infrastructure, and, as a running example of a data product, consider the task of building a service to suggest songs to users. This music suggestion service is an example of a *recommender system*, a popular class of data products which target content to individuals based on implicit (e.g., clicks) or explicit (e.g., ratings) feedback.

To begin making recommendations, we start with a *training dataset* that contains an existing set of user ratings for songs (e.g., songs that interns have manually labeled) and fit a statistical model to the training set. After this initial training, we iteratively improve the quality of the model as we collect more ratings.

There are numerous techniques for modeling this data; in this example, we consider widely used matrix factorization models [10]. At a high level, these models embed users and songs into a high-dimensional space of latent factors and use a distance metric between each as a proxy for similarity. More formally, these models *learn* a d-dimensional latent factor $w_u \in \mathbb{R}^d$ for each user u (collected into a matrix $W \in \mathbb{R}^{|\text{users}| \times d}$) and $x_i \in \mathbb{R}^d$ for each song i (and corresponding $X \in \mathbb{R}^{|\text{songs}| \times d}$). These latent factors encode information about unmeasured properties of the user (e.g., *DeadHead*) and song (e.g., *PartyAnthem*) and are learned by solving the following optimization problem:

$$\arg\min_{W,X} \lambda \left(||W||_2^2 + ||X||_2^2 \right) + \sum_{(u,i) \in \text{Obs}} (r_{ui} - w_u^T x_i)^2$$

Given the W and X matrices, we can calculate a user u's expected rating for a song i by appropriately projecting W and X to yield u's weights w_u and i's weights x_i, and taking their dot product:

$$\text{rating}(u, i) = w_u^T x_i$$

Therefore, an implementation of our data product has two natural phases. The first calculates the optimal W and X matrices containing factors for each user and item. The second uses W and X to make predictions for specific user-item pairs.

Similar to many companies providing actual data products, we could implement our song recommendation service by combining cluster compute frameworks with traditional data management and serving systems. For the training phase, we might compute W and X periodically (e.g., daily) using a large-scale cluster compute framework like Spark or GraphLab [9] based on a snapshot of the ratings logs stored in a distributed filesystem like HDFS [16]. In this architecture, models are trained on stale data and not updated in response to new user feedback until the next day.

For the serving phase, there are several options. The simplest strategy would precompute all predictions for every user and song combination and load these predictions into a lower-latency data store. While this approach hides the modeling complexity from the serving tier, it has the disadvantage of materializing potentially billions of predictions when only a small fraction will likely be required. Alternatively, a more sophisticated approach might load the latent factors in to a data management system and compute the predictions online in the application tier. Given the current lack of a general-purpose, scalable prediction service, this is likely to be an ad-hoc task that will be repeated for each data product.

2.1 Challenges and Opportunities

While the above straw-man proposal is a reasonable representation of how users today implement data products, there are several opportunities for improving the model management experience. Here, we highlight the challenges in managing a data product that are not addressed by either traditional offline analytics systems or online data management systems.

Low Latency: Because many data products are consumed by user-facing applications it is essential that they respond within the window of interactivity to prediction queries and quickly learn from new information. For example, a user listening to an online radio station expects their feedback to influence the next songs played by the music service. These demands for real-time responsiveness both in their ability to make predictions and learn from feedback are not well addressed by traditional cluster compute frameworks designed for scalable but batch-oriented machine learning. Additionally, while data management systems provide low latency query serving, they are not capable of retraining the underlying models.

Velox's approach: Velox provides low latency query serving by intelligently caching computation, scaling out model prediction and online training, and introducing new strategies for fast incremental model updates.

Large scale: With access to large amounts of data, the machine learning community has developed increasingly sophisticated techniques capable of modeling data at the granularity of individual entities. In our example recommender service, we learn factor representations for individual users and songs. Furthermore, these latent factor representations are interdependent: changes in the song factors affects the user factors. The size and interdependency of these models poses unique challenges to our ability to serve and maintain these models in a low latency environment.

Velox's approach: Velox addresses the challenge of scalable learning by leveraging existing cluster compute frameworks to initialize the model training process and infer global properties offline and then applies incremental maintenance strategies to efficiently update the model as more data is observed. To serve models at scale, Velox leverages distributed in memory storage and computation.

Model lifecycle management: Managing multiple models and identifying when models are underperforming or need to be retrained is a key challenge in the design of effective data products. For example, an advertising service may run a series of ad campaigns,

each with separate models over the same set of users. Alternatively, existing models may no longer adequately reflect the present state of the world. For example, a recommendation system that favors the songs in the Top 40 playlists may become increasingly less effective as new songs are introduced and the contents of the Top 40 evolves. Being able to identify when models need to be retrained, coordinating offline training, and updating the online models is essential to provide accurate predictions. Existing solutions typically bind together separate monitoring and management services with scripts to trigger retraining, often in an ad-hoc manner.

Velox's approach: Velox maintains statistics about model performance and version histories, enabling easier diagnostics of model quality regression and simple rollbacks to earlier model versions. In addition, Velox uses these statistics to automatically trigger offline retraining of models that are under performing and migrates those changes to the online serving environment.

Adaptive feedback: Because data products influence the actions of other services and ultimately the data that is collected, their decisions can affect their ability to learn in the future. For example, a recommendation system that only recommends sports articles may not collect enough information to learn about a user's preferences for articles on politics. While there are a range of techniques in the machine learning literature [14, 20] to address this feedback loop, these techniques must be able to modify the predictions served and observe the results, and thus run in the serving layer.

Velox's approach: Velox adopts bandit techniques [14] for controlling feedback that enable the system to optimize not only prediction accuracy but its ability to effectively learn the user models.

3. SYSTEM ARCHITECTURE

In response to the challenges presented in Section 2, we developed Velox, a system for serving, incrementally maintaining, and managing machine learning models at scale within the existing BDAS ecosystem. Velox allows BDAS users to build, maintain, and operate full, end-to-end data products. In this section, we outline the Velox architecture.

Velox consists of two primary architectural components. First, the **Velox model manager** orchestrates the computation and maintenance of a set of pre-declared machine learning models, incorporating feedback and new data, evaluating model performance, and retraining models as necessary. The manager performs fine-grained model updates but, to facilitate large scale model re-training, uses Spark for efficient batch computation.

Second, the **Velox model predictor** implements a low-latency, up-to-date prediction interface for end-users and other data product consumers. There are a range of pre-materialization strategies for model predictions, which we outline in detail in Section 5.

To persist models and training data, Velox uses a configurable storage backend. By default, Velox is configured to use Tachyon [13], a fault-tolerant, memory-optimized distributed storage system in BDAS. In our current architecture, both the model manager and predictor are deployed as a pair of co-located processes that are resident with each Tachyon worker process. When coupled with an intelligent routing policy, this design maximizes data locality.

Current modeling functionality: The current Velox implementation provides native support for a simple yet highly expressive family of personalized linear models that generalizes the matrix factorization model presented in Section 2. (We discuss extensions in Section 8.) Each model consists of a d-dimensional weight vector $w_u \in \mathbb{R}^d$ for each user u, and a feature function f which maps an input object (e.g., a song) into a d-dimensional feature space

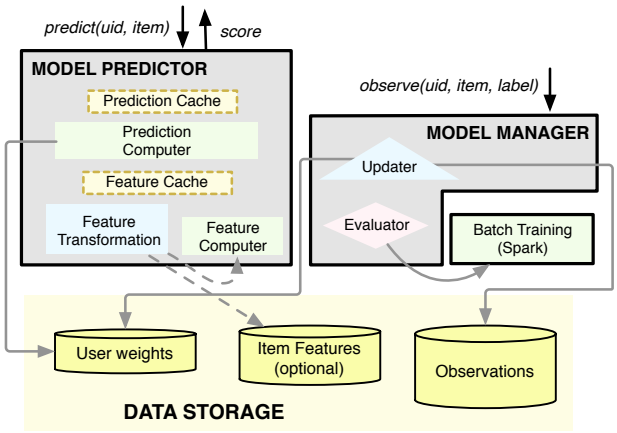

Figure 2: **Velox System Architecture** Two core components, the Model Predictor and Model Manager, orchestrate low latency and large-scale serving of data products computed (originally, in batch) by Spark and stored in a lightweight storage layer (e.g., Tachyon).

(e.g., its latent factor representation). A prediction is computed by evaluating:

$$\text{prediction}(u, x) = w_u^T f(x, \theta) \quad (1)$$

The feature parameters θ in conjunction with the feature function f enable this simple model to incorporate a wide range of models including support vector machines (SVMs) [5], deep neural networks [3], and the latent factor models used to build our song recommendation service.

Recommendation service behavior: In our music recommendation data product, the Velox prediction service (Section 5) computes predictions for a given user and item (or set of items) by reading the user model w_u and feature parameters θ from Tachyon and evaluating Eq. (1). The Velox model maintenance service (Section 4) updates the user-models w_u as new observations enter the system and evaluates the resulting model quality. When the model quality is determined to have degraded below some threshold, the maintenance service triggers Spark, the offline training component, to retrain the feature parameters θ. Spark consumes newly observed data from the storage layer, recomputes the user models and feature parameters, and writes the results back to Tachyon.

In the subsequent sections, we provide greater detail about the design of each component, the interfaces they expose, and some of the design decisions we have made in our early prototype. While our focus is on exposing these generalized linear models as data products to end-users, we describe the process of implementing additional models in Section 6.

4. MODEL MANAGEMENT

The model management component of Velox is responsible for orchestrating the model life-cycle including: collecting observation and model performance feedback, incrementally updating the user specific weights, continuous evaluation of model performance, and the offline retraining of feature parameters.

4.1 Feedback and Data Collection

As users interact with applications backed by Velox, the front-end applications can gather more data (both explicitly and implicitly) about a user's behavior. Velox exposes an *observation* interface to consume this new interaction data and update the user's model accordingly. To insert an observation about a user-item pair into

Velox, a front-end application calls observe (Listing 1), providing the user's ID, the item data (for feature extraction), and the correct label y for that item. In addition to being used to trigger online updates, the observation is written to Tachyon for use by Spark when retraining the model offline.

4.2 Offline + Online Learning

Learning in the Velox modeling framework consists of estimating the user specific weights w_u as well as the parameters θ of the feature function. To simultaneously enable low latency learning and personalization while also supporting sophisticated models we divide the learning process into online and offline phases. The offline phase adjusts the feature parameters θ and can run infrequently because the feature parameters capture aggregate properties of the data and therefore evolve slowly. The online phase exploits the independence of the user weights and the linear structure of Eq. (1) to permit lightweight conflict free per user updates.

The infrequent offline retraining phase leverages the bulk computation capabilities of large-scale cluster compute frameworks to recompute the feature parameters θ. The training procedure for computing the new feature parameters is defined as an opaque Spark UDF and depends on the current user weights and all the available training data. The result of offline training are new feature parameters as well as potentially updated user weights.

Because the offline phase modifies both the feature parameters and user weights it invalidates both prediction and feature caches. To alleviate some of the performance degradation resulting from invalidating both caches, the batch analytics system also computes all predictions and feature transformations that were cached at the time the batch computation was triggered. These are used to repopulate the caches when switching to the newly trained model. We intend to investigate further improvements in this process, as it is possible that the set of hot items may change as the retrained models redistribute the distribution of popularity among items.

The online learning phase runs continuously and adapts the user specific models w_u to observations as they arrive. While the precise form of the user model updates depends on the choice of error function (a configuration option) we restrict our attention to the widely used squared error (with L_2 regularization) in our initial prototype. As a consequence the updated value of w_u can be obtained analytically using the normal equations:

$$w_u \leftarrow (F(X,\theta)^T F(X,\theta) + \lambda I_n)^{-1} F(X,\theta)^T Y \qquad (2)$$

where $F(X,\theta) \in \mathbb{R}^{n_u \times d}$ is the matrix formed by evaluating the feature function on all n_u examples of training data obtained for that particular user and λ is a fixed regularization constant. While this step has cubic time complexity in the feature dimension d and linear time complexity in the number of examples n it can be maintained in time quadratic in d using the Sherman-Morrison formula for rank-one updates. Nonetheless using a naive implementation we are able to achieve (Figure 3) acceptable latency for a range of feature dimensions d on a real-world collaborative filtering task.

By updating the user weights online and the feature parameters offline, we are provide an *approximation* to continuously retraining the entire model. Moreover, while the feature parameters evolve slowly, they still change. By not continuously updating their value, we potentially introduce inaccuracy into the model. To assess the impact of the hybrid online + offline incremental strategy adopted by Velox, we evaluated the accuracy of Velox on the MovieLens10M dataset[1]. By initializing the latent features with 10 ratings from each user and then using an additional 7 ratings, we were able to achieve 1.6% improvement in prediction accuracy[2] by applying the online strategy. This is comparable to the 2.3% increase in accuracy achieved using full offline retraining.

We first used offline training to initialize the feature parameters θ on half of the data and then evaluated the prediction error of the proposed strategy on the remaining data. By using the Velox's incremental online updates to train on 70% of the remaining data, we were able to achieve a held out prediction error that is only slightly worse than complete retraining.

[1] http://grouplens.org/datasets/movielens

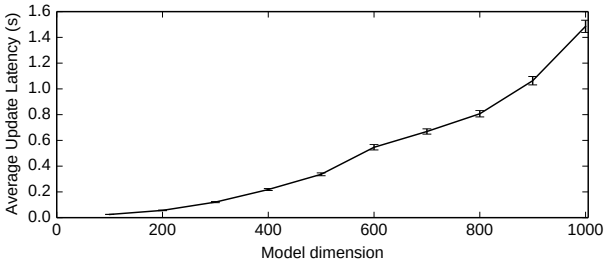

Figure 3: **Update latency vs model complexity** Average time to perform an online update to a user model as a function of the number of factors in the model. The results are averaged over 5000 updates of randomly selected users and items from the MovieLens 10M rating data set. Error bars represent 95% confidence intervals.

```
def predict(s: ModelSchema, uid: UUID, x: Data)
  : (Data, Double)
def topK(s: ModelSchema, uid: UUID, x: List[Data])
  : List[(Data, Double)]
def observe(uid: UUID, x: Data, y: Double)
```

Listing 1: **The Prediction and Observation API** These methods form the front-end API for a prediction and model management service (i.e., Velox).

4.3 Model Evaluation

Monitoring model performance is an essential part of any predictive service. Velox relies on model performance metrics to both aid administrators in managing deployed models and to identify when offline retraining of feature parameters is required. To assess model performance, Velox applies several strategies. First, Velox maintains running per-user aggregates of errors associated with each model. Second, Velox runs an additional cross-validation step during incremental user weight updates to assess generalization performance. Finally, when the topK prediction API is used, Velox employs bandit algorithms to collect a pool of validation data that is not influenced by the model. When the error rate on any of these metrics exceeds a pre-configured threshold, the model is retrained offline.

5. ONLINE PREDICTION SERVICE

The Velox prediction service exposes model predictions to other services and applications through a simple interface (Listing 1) The predict function serves point predictions for the provided user and item, returning the item and its predicted score. The topK function evaluates the best item from the provided set for the given uid. Support for the latter function is necessary for Velox to implement the bandit methods described later in this section and can be used to support pre-filtering items according to application level policies.

[2] Differences in accuracy on the MovieLens dataset are typically measured in small percentages.

Caching: The dominant expense when serving predictions in Velox is evaluating the feature function f. In particular, when f represents a materialized feature function (e.g., matrix factorization models), the distributed lookup of the latent factor in θ is the dominant cost. Alternatively, when f represents a computational feature function (e.g., a deep neural network) the computation becomes the dominant cost. These costs reflect two opportunities for optimization: caching the results of feature function evaluations and efficiently partitioning and replicating the materialized feature tables to limit remote data accesses. Velox performs both caching strategies in the Velox predictor process, corresponding to the *Feature Cache* in Figure 2. In addition, we can cache the final prediction for a given (user,item) pair, often useful for repeated calls to topK with overlapping itemsets, corresponding to the *Prediction Cache* in Figure 2.

To demonstrate the performance improvement that the prediction cache provides, we evaluated the prediction latency of computing topK for itemsets of varying size. We compare the prediction latency for the optimal case, when all predictions are cached (i.e., 100% cache hit rate) with the prediction latencies for models of several different sizes. As Figure 4 demonstrates, the relationship between itemset size and prediction latency grows linearly, which is to be expected. And as the model size grows (a simple proxy for the expense of computing a prediction, which is a product of both the prediction expense and the feature transformation expense), the benefits of caching grow.

To distribute the load across a Velox cluster and reduce network data transfer, we exploit the fact that every prediction is associated with a specific user and partition W, the user weight vectors table, by uid. We then deploy a routing protocol for incoming user requests to ensure that they are served by the node containing that user's model. This partitioning serves a dual purpose. It ensures that lookups into W can always be satisfied locally, and it provides a natural load-balancing scheme for distributing both serving load and the computational cost of online updates. This also has the beneficial side-effect that all writes — online updates to user weight vectors — are local.

When the feature function f is materialized as a pre-computed lookup table, the table is also partitioned across the cluster. Therefore, evaluating f may involve a data transfer from a remote machine containing the required item-features pair. However, while the number of items in the system may be large, item popularity often follows a Zipfian distribution [15]. Consequently, many items are not frequently accessed, and a small subset of items are accessed very often. This suggests that caching the hot items on each machine using a simple cache eviction strategy like LRU, will tend to have a high hit rate. Fortunately, because the materialized features for each item are only updated during the offline batch retraining, cached items are invalidated infrequently.

When the feature transformation f is computational, caching the results of computing the basis functions can provide similar benefits. For popular items, caching feature function evaluation reduces prediction latency by eliminating the time to compute the potentially expensive feature function, and reduces computational load on the serving machine, freeing resources for serving queries.

Bootstrapping: One of the key challenges in predictive services is how to model new users. In Velox, we currently adopt a simple heuristic in which new users are assigned a recent estimate of the average of the existing user weight vectors:

$$\bar{w}^T f(x, \theta) = \frac{1}{|\text{users}|} \sum_{u \in \text{users}} w_u^T f(x, \theta)$$

This also corresponds predicting the average score for all users.

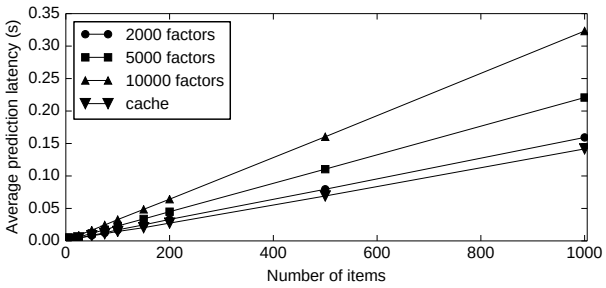

Figure 4: **Prediction latency vs model complexity** Single-node topK prediction latency for both cached and non-cached predictions for the Movie Lens 10M rating dataset, varying size of input set and dimension (d, or, factor). Results are averaged over 10,000 trials.

Bandits and Multiple Models: Model serving influences decisions that may, in turn, be used to train future models. This can lead to feedback loops. For example, a music recommendation service that only plays the current Top40 songs will never receive feedback from users indicating that others songs are preferable. To escape these feedback loops we rely on a form of the *contextual bandits algorithm* [14], a family of techniques developed to avoid these feedback loops. These techniques assign each item an *uncertainty* score in addition to its predicted store. The algorithm improves models greedily by reducing uncertainty about predictions. To reduce the total uncertainty in the model, the algorithm recommends the item with the best *potential* prediction score (i.e., the item with max sum of score and uncertainty) as opposed to recommending the item with the absolute best prediction score. When Velox observes the correct score for that recommendation and an online update is triggered, that update will reduce the uncertainty in the user weight vector more so than an observation about an item with less uncertainty. The bandits algorithms exploit the topK interface to select the item that has the highest potential predicted rating. For example, if Velox is unsure to what extent a user is a *DeadHead* it will occasionally select songs such as "New Potato Caboose" to evaluate this hypothesis even if those songs do not have the highest prediction score.

6. ADDING NEW MODELS

It is possible to express a wide range of models and machine learning techniques within Velox by defining new feature functions. To add a new model to Velox, a data scientist implements and uploads a new $\mathrm{VeloxModel}$ instance (Listing 2).

Shared state: Each $\mathrm{VeloxModel}$ may be instantiated with a vector, used to provide any global, immutable state (i.e., θ from Section 2) needed during the featurization process. For example, this state may be the parameters for a set of SVMs learned offline and used as the feature transformation function.

Feature transformations: The $\mathrm{VeloxModel}$ function $\mathrm{features}$ implements the feature transformation function. The $\mathrm{features}$ function may implement a computation on some input data, as is the case when the feature transformation is the computation of a set of basis functions. Alternatively, the $\mathrm{features}$ function may implement a lookup of the latent features in a table, similar to the table W used to store the user models. The implementor indicates which of these two strategies is used by explicitly specifying whether the features are materialized or are computed. Continuing with the ensemble of SVMs example, $\mathrm{features}$ would evaluate a set of SVMs with different parameters (stored in the member state) passed in on instance construction. We are investigating automatic ways of analyzing data

```
class VeloxModel:
  val name: String  // <- user provided
  val state: Vector // <- feature parameters
  val version: Int  // <- system provided
  def VeloxModel(state: Opt[Vector])
  // Feature Transformation Function
  def features(x: Data, materialized: Boolean)
      : Vector
  // Learning
  def retrain(f: (Data) => Vector,
              w: Table[String, Vector],
              newData: Seq[Data])
      : ((Data) => Vector, Table[String,Vector])
  // Quality Evaluation
  def loss(y: Label, yPredict: Label,
           x: Data, uid: UUID): Double
```

Listing 2: **The** VeloxModel **Interface** Developers can add new models and feature transformation functions to Velox by implementing this interface. Implementations specify how to featurize items, perform offline training, and how to evaluate model quality.

dependencies through techniques like UDF byte-code inspection.

Quality evaluation and model retraining: The user provides two functions, retrain and loss, that allow Velox to automatically detect when models are stale and retrain them. The loss is evaluated every time new data is observed (i.e., every time a user model is updated) and if the loss starts to increase faster than a threshold value, the model is detected as stale. Once a model has been detected as stale, Velox retrains the model offline using the cluster compute framework. retrain informs the cluster compute framework how to train this VeloxModel, as well as where to find and how to interpret the observation data needed for training. When offline training completes, Velox automatically instantiates a new VeloxModel and new W — incrementing the version — and transparently upgrades incoming prediction requests to be served with the newly trained user-models and VeloxModel.

7. RELATED WORK

Velox draws upon a range of related work from the intersection of database systems and complex analytics. We can broadly characterize this work as belonging to three major areas:

Predictive database systems: The past several years have seen several calls towards tight coupling of databases and predictive analytics. The Longview system [1] integrates predictive models as first-class entities in PostgreSQL and introduces a declarative language for model querying. Similarly, Bismarck [8] allows users to express complex analytics via common user-defined aggregates. A large body of work studies probabilistic databases, which provide first-class support for complex statistical models but, in turn, focus on modeling uncertainty in data [18, 19]. Commercially, the PMML markup language and implementations like Oryx[3] provide support for a subset of the data product concerns addressed by Velox. Our focus in Velox is to provide predictive analytics as required for modern data products in a large scale distributed setting. In doing so, we focus on user-specific personalization, online model training, and the challenges of feedback loops in modern predictive services.

In contrast with prior work on model management in database systems, which was largely concerned with managing deterministic metadata (such as schema) [4], Velox focuses on the use and management of statistical models from the domain of machine learning.

[3]https://github.com/cloudera/oryx

View materialization: The problem of maintaining models at scale can be viewed as an instance of complex materialized view maintenance. In particular, MauveDB exploits this connection in a single-node context, providing a range of materialization strategies for a set of *model-based views* including regression and Kalman filtering [7] but does not address latent feature models or personalized modeling. Similarly, Columbus [22] demonstrates the power of caching and model reuse in in-situ feature learning. We see considerable opportunity in further exploiting the literature on materialized view maintenance [6] in the model serving setting.

Distributed machine learning: There are a bevy of systems suitable for the performing batch-oriented complex analytics tasks [2], and a considerable amount of work implementing specific tasks. For example, Li et al. [12] explored a strategy for implementing a variant of SGD within the Spark cluster compute framework that could be used by Velox to improve offline training performance. Our focus is on leveraging these existing algorithms to provide better *online* predictive tasks. However, we aggressively exploit these systems' batch processing ability and large install bases in our solution.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced Velox, a system for performing machine learning model serving and model maintenance at scale. Velox leverages knowledge of prediction semantics to efficiently cache and replicate models across a cluster. Velox updates models to react to changing user patterns, automatically monitoring model quality and delegating offline retraining to existing cluster compute frameworks. In doing so, Velox fills a void in current production analytics pipelines, simplifying front-end applications by allowing them to consume predictions from automatically maintained complex statistical models.

We have completed an initial Velox prototype that exposes a RESTful client interface and integrates with existing BDAS components, relying on Spark and Tachyon for offline training and distributed data storage. By running tests against the MovieLens10M dataset we demonstrated that our early prototype performs well on basic serving and model update tasks. In addition we have evaluated our online incremental update strategy and demonstrated that it closely recovers the prediction accuracy of offline batch retraining.

We are actively pursuing several areas of further research within Velox. While we have chosen a fairly conservative modeling interface thus far, we are investigating alternative prediction and modeling APIs—in particular, their effect on more sophisticated query planning and materialization strategies. We plan to integrate and evaluate additional multi-armed bandit (i.e., multiple model) techniques from the machine learning literature (including their dynamic updates) as well as more efficient top-K support for our linear modeling tasks. Velox will be released as open source software, and we anticipate an alpha code release in early 2015.

Acknowledgments

The authors would like to thank Joseph M. Hellerstein, Tomer Kaftan, Henry Milner, Ion Stoica, Vikram Sreekanti, and the anonymous CIDR reviewers for their thoughtful feedback on this work. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, the NSF Graduate Research Fellowship (grant DGE-1106400), and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

9. REFERENCES

[1] M. Akdere et al. The case for predictive database systems: Opportunities and challenges. In *CIDR*, 2011.

[2] S. Babu and H. Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trends in Databases*, 5(1), 2013.

[3] Y. Bengio. Learning deep architectures for AI. *Found. Trends Mach. Learn.*, 2(1):1–127, Jan. 2009.

[4] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[5] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, June 1998.

[6] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[7] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[8] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.

[9] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[10] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, Aug. 2009.

[11] 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.

[12] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *EDBT*, 2013.

[13] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC*, 2014.

[14] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, 2010.

[15] R. Meka et al. Matrix completion from power-law distributed samples. In *NIPS*. 2009.

[16] K. Shvachko et al. The hadoop distributed file system. In *MSST*. IEEE, 2010.

[17] E. R. Sparks et al. MLI: An API for distributed machine learning. In *ICDM*, 2013.

[18] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[19] D. Z. Wang et al. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. 2008.

[20] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[21] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[22] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, 2014.