Continuous Deployment of Machine Learning Pipelines

ABSTRACT

Today machine learning is entering many business and scientific applications. The life cycle of machine learning applications consists of data preprocessing for transforming the raw data into features, training a model using the features, and deploying the model for answering prediction queries. In order to guarantee accurate predictions, one has to continuously monitor and update the deployed model and pipeline. Current deployment platforms update the model using online learning methods. When online learning alone is not adequate to guarantee the prediction accuracy, some deployment platforms provide a mechanism for automatic or manual retraining of the model. While the online training is fast, the retraining of the model is time-consuming and adds extra overhead and complexity to the process of deployment.

We propose a novel continuous deployment approach for updating the deployed model using a combination of the incoming real-time data and the historical data. We utilize fast sampling techniques to include the historical data in the training process, thus eliminating the need for retraining of deployed model. We also offer online statistics computation and materialization of the preprocessed features, which further reduces the total training and data preprocessing time. In our experiments, we design and deploy two pipelines and models to process two real-world datasets. The experiments show that the continuous deployment reduces the total training cost up to 13 times while providing the same level of quality when compared to the state-of-the-art deployment approaches.

1 INTRODUCTION

In machine learning applications, a pipeline, a series of complex data processing steps, processes a labeled training dataset and produces a machine learning model. The model then has to be deployed into a deployment platform where it answers prediction queries in real-time. To properly preprocess the prediction queries, typically the pipeline has to be deployed alongside the model.

A deployment platform must be robust, i.e., it should accommodate many different types of machine learning models and pipelines. Moreover, it has to be simple to tune. Finally, the platform must maintain the quality of the model by further training the deployed model when new training data becomes available.

Online deployment of machine learning models is one method for maintaining the quality of a deployed model. In the online deployment approach, the deployment platform utilizes online learning methods to further train the deployed model. In online learning, the model is updated based on the incoming training data. Online learning adapts the model to the new training data and provides an up-to-date model. However, purely online deployment approach degrades the quality over time, rendering it ineffective in many cases. In some applications that online learning has proven to be effective, to guarantee a high level of quality, one has to tune the online learning method to the specific use case [20, 21]. Thus, effective online deployment of machine learning models cannot provide robustness and simplicity.

To solve the problem of degrading model quality, periodical deployment approach is utilized. In the periodical deployment approach, the platform, in addition to utilizing simple online learning, periodically retrains the deployed model using the historical data. One of the challenges in many real-world use cases is the size of the training datasets. Typically, training datasets are extremely large and require hours or days of data preprocessing and training to result in a new model. Despite this drawback, in some applications, retraining the model is still critical, as even a small increase in the quality of the deployed model can have a large impact. For example, in the domain of ads click-through rate (CTR) prediction, even a 0.1% accuracy improvement yields hundreds of millions of dollars in revenue [19]. In the periodical deployment approach, while the model is being retrained, new prediction queries and training data are still arriving at the deployment platform. However, the platform has to answer the prediction queries using the currently deployed model. Moreover, the platform appends the new training data to the historical data. By the time the retraining process is over, enough training data is accumulated which requires the deployment platform to perform another retraining. As a result, the deployed model quickly becomes stale.

An efficient deployment platform, aside from being robust and easy to tune, must be able to maintain the quality of the deployed model without incurring the high training cost of the periodical deployment approach.

We propose a deployment platform, that completely eliminates the need for retraining, thus significantly reducing the training cost while achieving the same level of quality as the periodical deployment approach. Our deployment platform is also robust and accommodates many different types of machine learning models and pipelines. Lastly, the tuning process of our deployment approach is simple and requires minimal user interaction.

Our deployment platform continuously updates the model using a combination of the historical and incoming training data. Similar to existing deployment platforms, our platform also utilizes online learning methods to update the model based on the incoming training data. However, instead of periodical retraining, our deployment platform performs regular updates to the model based on samples of the historical data. Our deployment platform offers the following two optimizations.

Proactive training. Proactive training is the process of utilizing samples of the historical data to update the deployed model. First, the deployment platform processes a given sample using the pipeline, then it computes a partial gradient and updates the deployed model based on the partial gradient. The updated model is immediately ready for answering prediction queries. Our experiments show that proactive training reduces the training time by an order of magnitude while providing the same level of quality when compared to the periodical deployment approach.

Online Statistics Computation and Feature Materialization. Before updating the model using proactive training, the pipeline has to preprocess the training data. Every component of the pipeline needs to scan the data, updates the statistics (for example the mean and the standard deviation of the standard scaler component), and finally transform the data. Computing these statistics and transforming the data are time consuming processes. Aside from the proactive training, our deployment platform also employs online learning methods to update the model in real-time. During the online learning, we compute the required statistics and transform the data. The deployment platform stores the updated statistics for every pipeline

component and materializes the transformed features by storing them in memory or disk. By reusing the computed statistics and the materialized features during the proactive training, we eliminate the data preprocessing steps of the pipeline and further decrease the proactive training time.

In summary, our contributions are:

- A platform for continuously training deployed machine learning models and pipelines that adapts to the changes in the incoming data. The platform accommodates different types of machine learning models and pipelines. In our experiments, we design and deploy two different machine learning pipelines.
- Proactive training of the deployed models and pipelines that frequently updates the model using samples of the historical data which guarantees high-quality models while completely eliminating the need for periodical retraining.
- Efficient pipeline processing and model training by online statistics computation and feature materialization, thus guaranteeing the availability of up-to-date models for answering prediction queries.

The rest of this paper is organized as follows: In Section 2, we describe the details of the optimization strategy we utilize for our continuous deployment platform. Section 3 describes the details of our continuous training approach. In Section 4, we introduce the architecture of our deployment platform. In Section 5, we evaluate the performance of our continuous deployment platform. Section 6 discusses the related work. Finally, Section 7 presents our conclusion and the future work.

2 BACKGROUND

To continuously train the deployed model, we rely on computing partial updates based on the current parameters of the model and a combination of the incoming and existing data. To compute the partial updates, we utilize Stochastic Gradient Descent (SGD) [32]. SGD has several parameters and in order to work effectively, they have to be tuned. In this section, we describe the details of SGD and its parameters and discuss the effect of the parameters on training machine learning models.

2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization strategy utilized by many machine learning algorithms for training a model. SGD is an iterative optimization technique where in every iteration, one data point or a sample of the data points is utilized to update the model. SGD is suitable for large datasets as it does not require scanning the entire data in every iteration [5]. SGD is also suitable for online learning scenarios, where new training data becomes available one at a time. Many different machine learning tasks such as classification [21, 32], clustering [6], and matrix factorization [13, 17] utilize SGD in training models. SGD is also the most common optimization strategy for training neural networks on large datasets [12]. Prominent applications of SGD in neural networks are the work of Google Deepmind team that managed to train neural networks that defeat humans in the game of Go [27] and mastering Atari games [22].

To explain the details of SGD, we describe how it is utilized to train a logistic regression model. In logistic regression, the goal is to find the weight vector (w) that maximizes the conditional likelihood of labels (y) based on the given data (x) in the training dataset:

$$w^* = \underset{w}{\operatorname{argmax}} \sum_{i=1}^{N} ln(P(y^i|x^i,w))$$

where N is the size of the training dataset. To utilize SGD for finding the optimal w, we start from initial random weights. Then in every iteration, we update the weights based on the gradient of the loss function:

$$w^{t+1} = w^t + \eta \sum_{i \in S} x^i (y^i - \hat{P}(Y^i = 1|x^i w))$$

where η is the learning rate parameter and S is the random sample in the current iteration. The algorithm continues until convergence, i.e., when the weight vector does not change after an iteration.

Learning Rate. An important parameter of stochastic gradient descent is the learning rate. The learning rate controls the degree of change in the weights during every iteration. The most trivial approach for tuning the learning rate is to initialize it to a small value and after every iteration decrease the value by a small factor. However, in complex and high-dimensional problems, the simple tuning approach is ineffective [25]. Adaptive learning rate methods such as Momentum [24], Adam [16], Rmsprop [29], and AdaDelta [31] have been proposed. These methods adaptively adjust the learning rate in every iteration to speed up the convergence rate. Moreover, some of the learning rate adaptation methods perform per coordinate modification, i.e., every parameter of the model weight vector is adjusted separately from the others [16, 29, 31]. In many high-dimensional problems, the parameters of the weight vector do not have the same level of importance, therefore each parameter must be treated differently during the training process.

Sample Size. Another parameter of stochastic gradient descent is the sample size. Given proper learning rate tuning mechanism, SGD eventually converges to a solution regardless of the sample size. However, the sample size can greatly affect the time that is required to converge. Two extremes of the sample size are 1 (every iteration considers 1 data item) and N (similar to batch gradient descent, every iteration scans the entire dataset). Setting the sample size to 1 increases the model update frequency but results in noisy updates. Therefore, more iterations are required for the model to converge. Using the entire data in every iteration leads to more stable updates. As a result, the model training process requires fewer iterations to converge. However, because of the size of the data, individual iterations require more time to complete. A common approach is mini-batch gradient descent. In mini-batch gradient descent, the sample size is selected in such a way that each iteration is fast. Moreover, the training process requires fewer iterations to converge.

Distributed SGD. To efficiently train machine learning models on large datasets, one has to employ scalable training algorithms. SGD inherently works well with large datasets because it does not need to scan every data point during every iteration. However, SGD has to perform many iterations to converge. To decrease the execution time, one can distribute the large dataset among multiple nodes. During the training, each node computes a partial gradient on a subset of the data in parallel. After this step, all the partial gradients are combined to compute the final gradient. Distributed SGD significantly reduces the time for executing individual iterations, which results in a reduction in the overall training time.

3 CONTINUOUS TRAINING APPROACH

In this section, first, we describe how we utilize the properties of the stochastic gradient descent to implement the proactive training. Next, we describe the details of the online statistics computation and feature materialization. Finally, we demonstrate how our deployment approach improves the existing process of deployment and maintains the quality of the deployed model.

3.1 Proactive Training

Proactive training is a replacement for the periodical retraining of the deployed model. Typically, the retraining is triggered when a condition is met, e.g., the quality of the model drops below a certain value. Contrary to the periodical training, in proactive training, the platform continuously update the deployed model using the historical data.

We take advantage of the iterative nature of SGD in the design of the proactive training. The input to each iteration of SGD is the current weight parameters of the model, a sample of the data points, and an objective function. In proactive training, we execute iterations of mini-batch SGD on the deployed model. To execute the proactive training, the deployment platform first samples the historical data. Then, the platform transforms the data into a set of features using the deployed pipeline. Next, the proactive trainer utilizes the transformed features to compute the gradient of the objective function. Finally, the deployment platform updates the deployed model using the computed gradient.

The learning rate parameter of SGD has a significant impact on the proactive training. To effectively update the deployed model, one has to tune the learning rate. Similar to the offline SGD, using a constant or decreasing value for the learning rate results in suboptimal training. Adaptive learning rate methods work well in a dynamic environment where the distribution of the data may change over time [31]. Therefore, in proactive training, instead of using simple learning rate tuning mechanisms, we utilize the more advanced learning rate adaptation methods. The performance of the different learning rate adaptation techniques varies across different datasets. To choose the most promising adaptation technique, we rely on hyperparameter tuning during the initial model training on the historical dataset [4]. After the initial training, the deployment platform selects the same learning rate adaptation technique for the proactive training. Our experiments in Section 5 show that selecting the adaptation technique based on initial training results in a model with the highest quality during the proactive training.

The proactive training aims to adapt the deployed model to the recent data items. As a result, when sampling the historical data, one has to consider the effect of the sample on the deployed model. In Section 4, we explain the different sampling strategies.

3.2 Online Statistics Computation and Feature Materialization

Before applying the proactive training, the deployment platform needs to transform the data using the deployed pipeline. Some components of the machine learning pipeline, such as the standard scaler or the one-hot encoder, require statistics over the dataset to be calculated before they process the data. Computing these statistics require scans of the data. In our deployment platform, we utilize online training as well as proactive training. During the online update of the deployed model, we compute all the necessary statistics for every component. Online computation of the required statistics eliminates the need to recompute the same statistics during the proactive training.

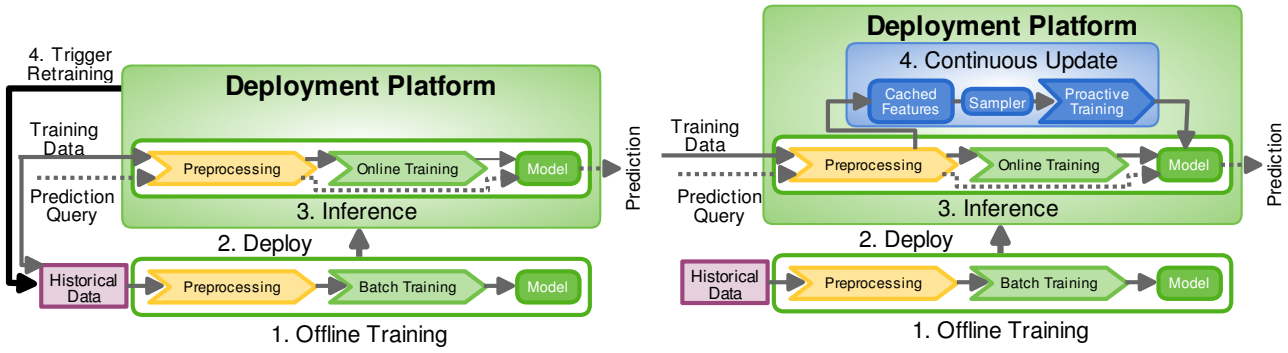
Moreover, during the online learning, the deployed pipeline transforms the incoming data to a set of features before updating the model. Given enough storage space, our deployment platform first assigns timestamps and then materializes the preprocessed features by storing them in a cache. Therefore, while performing the proactive training, instead of sampling from the raw historical data, the deployment platform samples the features directly from the cache. Materializing the features eliminates the data preprocessing part of the pipeline during the proactive training which significantly reduces the total training time for the proactive training.

Dynamic model size. Some components of the machine learning pipeline generate new features. For example, one-hot encoding and data bucketization, both may generate new features after processing new training data. When such components exist in the deployed pipeline, the deployment platform keeps track of the number of features after the data preprocessing. When the pipeline generates new features, the deployment platform adjusts the size of the deployed model.

3.3 Improved Deployment Process

Figure 1 shows the differences in deployment processes of the periodical and continuous training approaches. Figure 1a shows the process of the periodical deployment approach. The process starts with an offline training (1). During the offline training, a user designs a machine learning pipeline that consists of several data and feature preprocessing steps. After the data preprocessing step, the user trains a machine learning model by utilizing a batch training algorithm. During the deployment step, the user deploys the model and the pipeline into the deployment platform (2). To perform inference, the deployment platform directs the incoming prediction queries through the preprocessing steps before utilizing the model to predict the label (3). During the online update step, the deployment platform directs the training data through the preprocessing steps of the pipeline and then, using an online training algorithm, the platform updates the model. Finally, the deployment platform accommodates periodical retraining of the pipeline by either automatically triggering a batch training or prompting the user to train and redeploy a new model to the deployment platform (4). During the periodical retraining, the deployment platform has to disable the online updating of the model.

Figure 1b shows how our continuous training approach improves the existing deployment process. Similar to the current deployment process, a user first trains a pipeline (1) and deploy it into the deployment platform (2). The deployment platform utilizes the deployed pipeline and model to answer prediction queries and update the model using the incoming training data (3). After transforming the incoming data into a set of features, the deployment platform stores the transformed features inside a cache. During the proactive training, the platform samples the materialized features and computes the gradient using the sample. Finally, the platform updates the deployed model using the gradient (4). In the new deployment approach, the platform continuously updates the pipeline and the deployed model



(a) Periodical deployment of machine learning pipelines

(b) Continuous deployment of machine learning applications

Figure 1: Machine Learning Deployment Platforms

without requiring a full retraining over the historical data. As a result, the deployment platform ensures the model is always up-to-date.

4 DEPLOYMENT PLATFORM

Our proposed deployment platform comprises of five main components: pipeline manager, data manager, scheduler, proactive trainer, and execution engine. Figure 2 gives an overview of the architecture of our platform and the interactions among its components. At the center of the deployment platform is the pipeline manager. The pipeline manager monitors the deployed pipeline and model, manages the processing of the training data and prediction queries, and enables the continuous update of the deployed model. The data manager and the scheduler enable the pipeline manager to perform proactive training. The proactive trainer component manages the execution of the iterations of SGD on the deployed model. The execution engine is responsible for executing the actual data transformation and model training components of the pipeline.

4.1 Scheduler

The scheduler is responsible for scheduling the proactive training. The scheduler instructs the pipeline manager when to execute the proactive training. The scheduler accommodates two types of scheduling mechanisms, namely, *static* and *dynamic*.

The static scheduling utilizes a user-defined parameter that specifies the interval between executions of the proactive training. This is a simple mechanism for use cases that require constant updates to the deployed model (for example, every minute). The dynamic scheduling tunes the scheduling interval based on the rate of the incoming prediction, prediction latency, and the execution time of the proactive training. The scheduler uses the following formula to compute the time when to execute the next proactive training:

$$T' = S * T * pr * pl$$

where T' indicates the time in seconds when the next proactive training is scheduled to execute, T is the execution time of the last proactive training, pl is the average prediction latency, and pr is the average number of prediction queries per second. S is the slack parameter. Slack is a user-defined parameter to hint the scheduler about the possibility of surges in the incoming prediction queries and training data. During a proactive training, a certain number of predictions queries arrive at the platform ($T * pr$) which requires $T * pr * pl$ seconds to be processed. The scheduler must guarantee that the deployment platform answers all the queries before executing the next proactive training ($T' > T * pr * pl$). A large slack value (≥ 2) results in a larger scheduling interval, thus allocating most of the resources of the deployment platform to the query answering component. A small slack value ($2 \geq S \geq 1$) results in smaller scheduling intervals. As a result, the deployment platform allocates more resources for training the model. A slack value of less than 1 increases the latency of prediction query answering. The scheduler computes the value of T for every proactive training. The deployment platform provides the scheduler with the values of pr and pl.

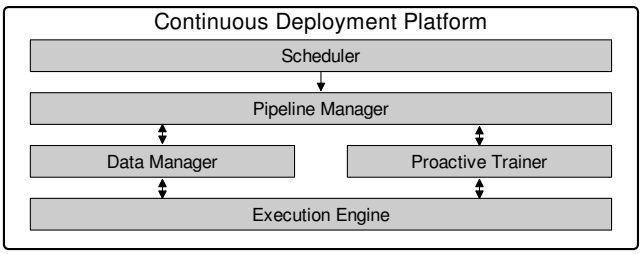


Figure 2: Architecture of the Continuous Deployment Platform

4.2 Data Manager

The data manager component is responsible for storage of historical data and materialized features, receiving the incoming training data, and providing the pipeline manager with samples of the historical data.

The data manager has four tasks. It appends the incoming training data to the historical dataset. It forwards the incoming training data and prediction queries to the pipeline manager for further processing. After the pipeline manager transforms the data into features, the data manager stores the transformed features. Finally, upon the request of the pipeline manager, the data manager samples the historical data for proactive training.

If the feature materialization optimization is enabled, the data manager stores the transformed features in the cache. During the sampling operation, instead of sampling from the raw historical data, the data manager samples from the transformed features directly.

The data manager provides three sampling approaches, namely, uniform, time-based, and window-based sampling. The uniform sampling provides a random sample from the entire historical data where every data point has the same probability of being sampled. Time-based sampling assigns weights to every data point based on their timestamp such that recent items have a higher probability of being sampled. Window-based sampling is similar to the uniform sampling, but instead of sampling from the entire historical data, the data manager samples the data from a given time range. In many real-world use cases (e.g., e-commerce and online advertising), the deployed model should adapt to the more recent data. Therefore, the time-based and window-based sampling provide more appropriate samples for the training data. However, in some use cases, the incoming training data is not time-dependent (e.g., image classification of objects). In these scenarios, window-based sampling fails to provide a non-biased sample, since it only samples from the recent data. In Section 5, we evaluate the quality of the deployed model when different sampling techniques are utilized. We show that the time-based sampling results in a model with better quality.

To increase the performance of the sampling operation, we utilize a data partitioning technique. Upon the arrival of new training data, the data manager assembles a partition of the data and creates an index for the partition using the timestamps of the data inside the partition. As a result, during the sampling operation, the data manager can access the data for a specific interval quickly. To further speed up the sampling, the data manager randomly samples the partitions instead of individual data points. The data manager combines the sampled partitions and sends the final data to the pipeline manager.

The data manager also allows for new training datasets to be registered while the model is being served. The new dataset is merged with the existing historical data and immediately becomes available for proactive training.

4.3 Pipeline Manager

The pipeline manager is the main component of the platform. It loads the pipeline and the trained model, transforms the data into features using the pipeline, enables the execution of the proactive training, and exposes the deployed model to answer prediction queries.

During the online training, when new training data becomes available, the pipeline manager transforms the data using the pipeline and updates the model. The pipeline manager also updates and stores the statistics of the pipeline components. If the feature materialization optimization is enabled, the pipeline manager sends the transformed features to the data manager.

The scheduler component informs the pipeline manager to execute proactive training. Then, the pipeline manager requests the data manager to provide it with a sample of the historical data for the next proactive training. The pipeline manager provides the proactive trainer with the current model parameter and the sample of the historical data. Once the proactive training is over, the pipeline manager receives the updated model.

The data manager also forwards the prediction queries to the pipeline manager. Similar to the training data, the pipeline manager sends the prediction queries through the pipeline to perform the necessary data preprocessing. Using the same pipeline to process both the training data and the prediction queries guarantees that the same set of transformations are applied to both types of data. As a result, the pipeline manager prevents inconsistencies between training and inference that is a common problem in the deployment of machine learning pipelines [3]. After preprocessing the prediction queries, the pipeline manager uses the deployed model to make predictions.

4.4 Proactive Trainer

The proactive trainer is responsible for training the deployed model by executing iterations of SGD. In the training process, the proactive trainer receives a training dataset (sample of the historical data) and the current model parameters from the pipeline manager. Then, the proactive trainer performs one iteration of SGD and returns the updated model to the pipeline manager. The proactive trainer utilizes advanced learning rate adaptation techniques to dynamically adjust the learning rate parameter when training the model. Although individual proactive training instances are independent of each other, the component must store the information required by the learning rate adaptation technique. Lastly, the proactive trainer executes the SGD logic on the execution engine. Therefore, to switch the execution engine, the proactive trainer must provide a new implementation of the SGD logic.

4.5 Execution Engine

The execution engine is responsible for executing the SGD and the prediction answering logic. In our deployment platform, any data processing platform capable of processing data both in batch mode (for proactive training) and streaming mode (online learning and answering prediction queries) is a suitable execution engine. Platforms such as Apache Spark [30], Apache Flink [7], and Google-DataFlow [2] are distributed data processing platforms that support both stream and batch data processing.

5 EVALUATION

To evaluate the performance of our deployment platform, we perform several experiments. Our main goal is to show that the continuous deployment approach maintains the quality of the deployed model while reducing the total training time. Specifically, we answer the following questions:

1. How does our continuous deployment approach perform in comparison to online and periodical deployment approaches with regards to model quality and training time?
2. What are the effects of the learning rate adaptation method, the regularization parameter, and the sampling strategy on the continuous deployment?
3. What are the effects of online statistics computation and materialization of preprocessed features on the training time?

To that end, we first design two pipelines each processing one real-world dataset. Then, we deploy the pipelines using different deployment approaches.

5.1 Setup

Pipelines. We design two pipelines for all the experiments.

URL pipeline. The URL pipeline processes the URL dataset for classifying URLs, gathered over a 121 days period, into malicious and legitimate groups [20]. The pipeline consists of 5 components: input parser, missing value imputer, standard scaler, feature hasher, and an SVM model. To evaluate the SVM model, we compute the misclassification rate on the unseen data.

Taxi Pipeline. The Taxi pipeline processes the New York taxi trip dataset and predicts the trip duration of every taxi ride [8]. The pipeline consists of 5 components: input parser, feature extractor, anomaly detector, standard scaler, and a Linear Regression model.

We design the pipeline based on the solutions of the top scorers of the New York City (NYC) Taxi Trip Duration Kaggle competition[1]. The input parser computes the actual trip duration by first extracting the pickup and drop off time fields from the input records and calculating the difference (in seconds) between the two values. The feature extractor computes the haversine distance[2], the bearing[3], the hour of the day, and the day of the week from the input records. Finally, the anomaly detector filters the trips that are longer than 22 hours, smaller than 10 seconds, or the trips that have a total distance of zero (the car never moved). To evaluate the model, we use the Root Mean Squared Logarithmic Error (RMSLE) measure. RMSLE is also the chosen error metric for the NYC Taxi Trip Duration Kaggle competition.

Deployment Environment. We deploy the URL pipeline on a single laptop running a macOS High Sierra 10.13.4 with 2,2 GHz Intel Core i7 and 16 GB of RAM and the Taxi pipeline on a cluster of 21 machines (Intel Xeon 2.4 GHz 16 cores, 28 GB of dedicated RAM per node). In our current prototype, we are using Apache Spark 2.2 as the execution engine. The data manager component utilizes the Hadoop Distributed File System (HDFS) 2.7.1 for storing the historical data [26]. We leverage some of the components of the machine learning library in Spark to implement the SGD logic. Moreover, we utilize the caching mechanism of the Apache Spark for materializing the transformed features.

Experiment and Deployment Process. Table 1 describes the details of the datasets such as the size of the raw data for the initial training, and the amount of data for the prediction queries and further training after deployment. For the URL pipeline, we first train a model on the first day of the data (day 0). For the Taxi pipeline, we train a model using the data from January 2015. For both datasets, since the entire data fits in the memory of the computing nodes, we use batch gradient descent (sampling ratio of 1.0) during the initial training. We then deploy the models (and the pipelines). We use the remaining data for sending prediction queries and further training of the deployed models.

For experiments that compare the quality of the deployed model, we utilize the prediction queries to compute the cumulative prequential error rate of the deployed models over time [11]. For experiments that capture the cost of the deployment, we measure the time the platforms spend in updating the model, performing retraining, and answering prediction queries.

The URL dataset does not have timestamps. Therefore, we divide every day of the data into micro-batches of 1 minute which results in 12000 micro-batches. We sequentially send the micro-batches to the deployment platform. The deployment platform first uses the micro-batch for prequential evaluation and then updates the deployed model. The Taxi dataset includes timestamps. In our experiments, each micro-batch of the Taxi dataset contains one hour of the data. The micro-batches are sent in order of the timestamps (from 2015-Feb-01 00:00 to 2016-Jun-30 24:00, an 18 months period) to the deployment platform.

5.2 Experiment 1: Deployment Approaches

In this experiment, we investigate the effect of our continuous deployment approach on model quality and the total training time. We use 3 different deployment approaches.

[1]https://www.kaggle.com/c/nyc-taxi-trip-duration/
[2]https://en.wikipedia.org/wiki/Haversine_formula
[3]https://en.wikipedia.org/wiki/Bearing_(navigation)

Dataset	size	# instances	Initial	Deployment
URL	2.1 GB	2.4 M	Day 0	Day 1-120
Taxi	42 GB	280 M	Jan15	Feb15 to Jun16

Table 1: Description of Datasets. The Initial and Deployment columns indicate the amount of data used during the initial model training and the deployment phase (prediction queries and further training data)

- Online: deploy the pipeline, then utilize online training (gradient descent with a sample size of 1 data point) method for updating the deployed model.
- Periodical: deploy the pipeline, then periodically retrain the deployed model.
- Continuous: deploy the pipeline, then continuously update the deployed model using our platform.

The periodical deployment initiates a full retraining every 10 days and every month for URL and Taxi pipelines, respectively. Since the rate of the incoming training and prediction queries are known, we use static scheduling for the proactive training. Based on the size and rate of the data, our deployment platform executes the proactive training every 5 minutes and 5 hours for the URL and Taxi pipelines, respectively. To improve the performance of the periodical deployment, we utilize the warm starting technique, used in the TFX framework [3]. In warm starting, each periodical training uses the existing parameters such as the pipeline statistics (e.g., standard scaler), model weights, and learning rate adaptation parameters (e.g., the average of past gradients used in Adadelta, Adam, and Rmsprop) when training new models.

Model Quality. Figure 3 (a) and (c) show the cumulative error rate over time for the different deployment approaches. For both datasets, the continuous and the periodical deployment result in a lower error rate than the online deployment. Online deployment visits every incoming training data point only once. As a result, the model updates are more prone to noise. This results in a higher error rate than the continuous and periodical deployment. In Figure 3 (a), during the first 110 days of the deployment, the continuous deployment has a lower error rate than the periodical deployment. Only after the final retraining, the periodical deployment slightly outperforms the continuous deployment. However, from the start to the end of the deployment process, the continuous deployment improves the average error rate by 0.3% and 1.5% over the periodical and online deployment, respectively. In Figure 3 (c), for the Taxi dataset, the continuous deployment always attains a smaller error rate than the periodical deployment. Overall, the continuous deployment improves the error rate by 0.05% and 0.1% over the periodical and online deployment, respectively.

When compared to the online deployment, periodical deployment slightly decreases the error rate after every retraining. However, between every retraining, the platform updates the model using online learning. This contributes to the higher error rate than the continuous deployment, where the platform continuously trains the deployed model using samples of the historical data.

In Figure 3 (b) and (d), we report the cumulative cost over time for every deployment platform. We define the deployment cost as the total time spent in data preprocessing, model training, and performing prediction. For the URL dataset (Figure 3 (b)), online deployment has the smallest cost (around 34 minutes) as it only scans each data point once (around 2.4 million scans). The continuous deployment approach scans 45 million data points. However, the total cost at

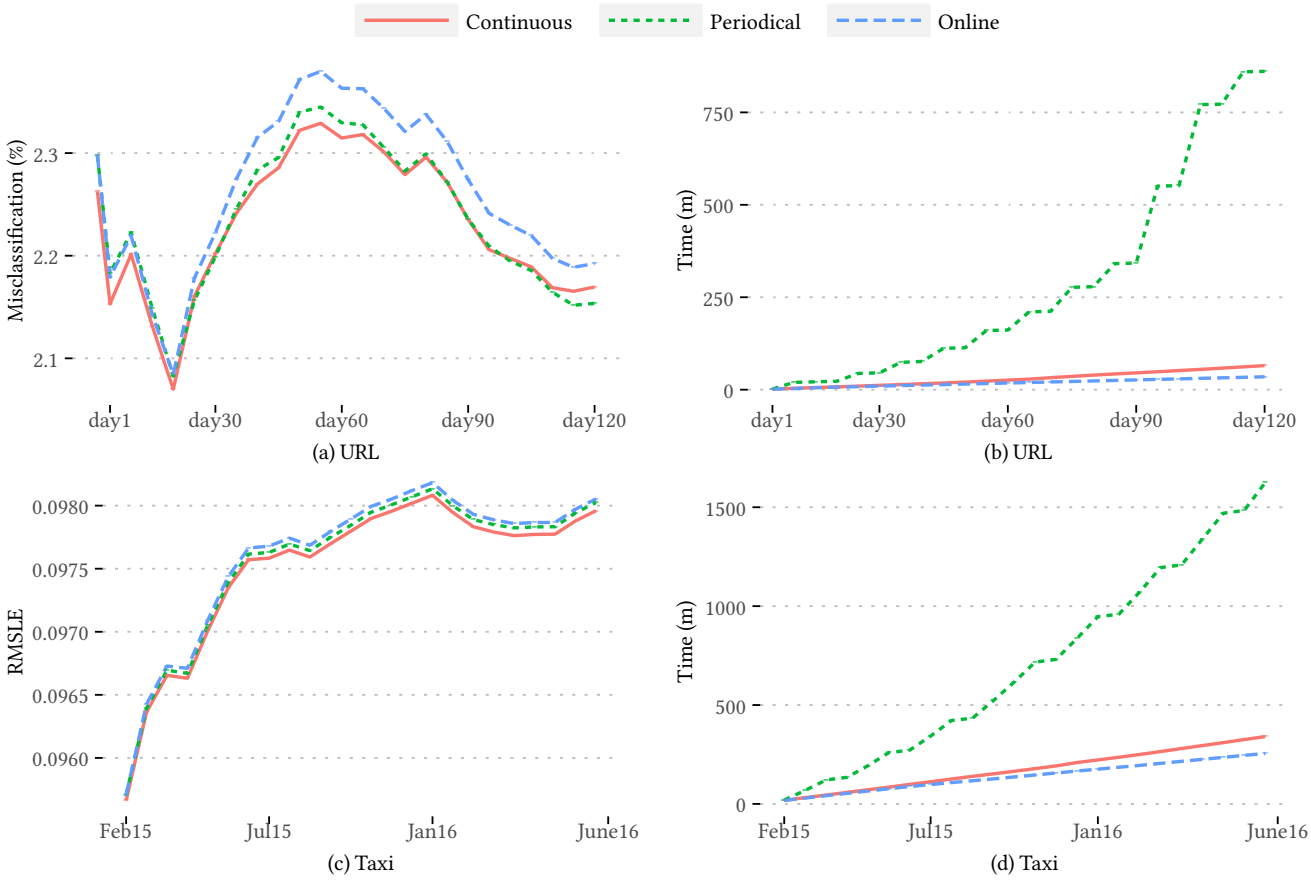


Figure 3: Model Quality and Training cost for different deployment approaches

the end of the deployment is only two times larger than the online deployment approach (around 65 minutes). Because of the online statistics computation and the feature materialization optimizations, a large part of the data preprocessing time is avoided. For the periodical deployment approach, the cumulative deployment cost starts similar to the online deployment approach. However, after every offline retraining, the deployment cost substantially increases. At the end of the deployment process, the total cost for the periodical deployment is more than 850 minutes which is 13 times more than the total cost of the continuous deployment approach. Each data point in the URL dataset has more than 3 million features. Therefore, the convergence time for each retraining is very high. The high data-dimensionality and repeated data preprocessing contribute to the large deployment cost of the periodical deployment.

For the Taxi dataset (Figure 3 (d)), the cost of online, continuous, and periodical deployments are 250, 340, and 1600 minutes, respectively. Similar to the URL dataset, continuous deployment only adds a small overhead to the deployment cost when compared with the online deployment. Contrary to the URL dataset, the feature size of the Taxi dataset is 11. Therefore, offline retraining converges faster to a solution. As a result, for the Taxi dataset, the cost of the periodical deployment is 5 times larger than the continuous deployment (instead of 13 times for URL dataset).

5.3 Experiment 2: System Tuning

In this experiment, we investigate the effect of different parameters on the quality of the models after deployment. As described in Section 3.1, proactive training is an extension of the stochastic gradient descent to the deployment phase. Therefore, we expect the set of hyperparameters with best performance during the initial training also performs the best during the deployment phase.

Proactive Training Parameters. Stochastic gradient descent is heavily dependent on the choice of learning rate and the regularization parameter. To find the best set of hyperparameters for the initial training, we perform a grid search. We use advanced learning rate adaptation techniques (Adam, Adadelta, and Rmsprop) for both initial and proactive training. For each dataset, we divide the initial data (from Table 1) into a training and evaluation set. For each configuration, we first train a model using the training set and then evaluate the model using the evaluation set. Table 2 shows the result of the hyperparameter tuning for every pipeline. For the URL dataset, Adam with regularization parameter $1E$-3 yields the model with the lowest error rate. The Taxi dataset is less complex than the URL dataset and has a smaller number of feature dimensions. As a result, the choice of different hyperparameter does not have a large impact on the quality of the model. The Rmsprop adaptation technique with

	URL			Taxi		
Adaptation	1E-2	1E-3	1E-4	1E-2	1E-3	1E-4
Adam	0.030	**0.026**	0.035	0.09553	0.09551	**0.09551**
RMSProp	0.030	**0.027**	0.034	0.09552	0.09552	**0.09550**
Adadelta	0.029	**0.028**	0.034	**0.09609**	0.09610	0.09619

Table 2: Hyperparameter tuning during initial training (bold numbers show the best results for each adaptation techniques)

regularization parameter of $1E$-4 results in a slightly better model than the other configurations.

After the initial training, for every configuration, we deploy the model and use 10 % of the remaining data to evaluate the model after deployment. Figure 4 shows the results of the different hyperparameter configurations on the deployed model. To make the deployment figure more readable, we avoid displaying the result of every possible combination of hyperparameters and only show the result of the best configuration for each learning rate adaptation technique. For the URL dataset, similar to the initial training, Adam with regularization parameter $1E$-3 results in the best model. For the Taxi dataset, we observe a similar behavior to the initial training where different configurations do not have a significant impact on the quality of the deployed model.

This experiment confirms that the effect of the hyperparameters (learning rate and regularization) during the initial and proactive training are the same. Therefore, we tune the parameters of the proactive training based on the result of the hyperparameter search during the initial training.

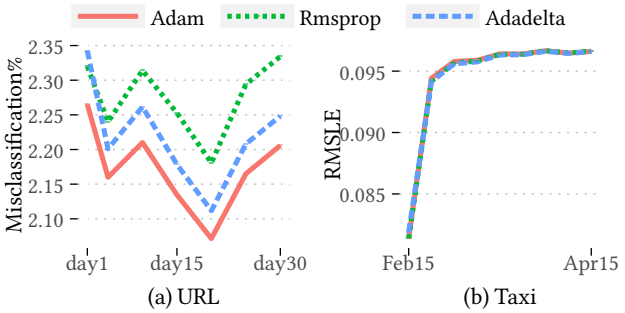


Figure 4: Result of hyperparameter tuning during the deployment

Sampling Methods. The choice of the sampling strategy also affects the proactive training. Each instance of the proactive training updates the deployed model using the provided sample. Therefore, the quality of the model after an update is directly related to the quality of the sample. We evaluate the effect of three different sampling strategies, namely, time-based, window-based, and uniform, on the quality of the deployed model. The sample size is similar to the sample size during the initial training ($16k$ and $1M$ for URL and Taxi data, respectively). Figure 5 shows the effect of different sampling strategies on the quality of the deployed model. For the URL dataset, time-based sampling improves the average error rate by 0.5% and 0.9% over the window-based and uniform sampling, respectively. As new features are added to the URL dataset over time, the underlying characteristics of the dataset gradually change [20]. A time-based sampling approach is more likely to select the recent items for the proactive training. As a result, the deployed model performs better on the incoming prediction queries. The underlying characteristics of the Taxi dataset is known to remain static over time. As a result, we observe that different sampling strategies have the same effect on the quality of the deployed model.

Our experiments show that for datasets that gradually change over time, time-based sampling outperforms other sampling strategies. Moreover, time-based sampling performs similarly to window-based and uniform sampling for datasets with stationary distributions.

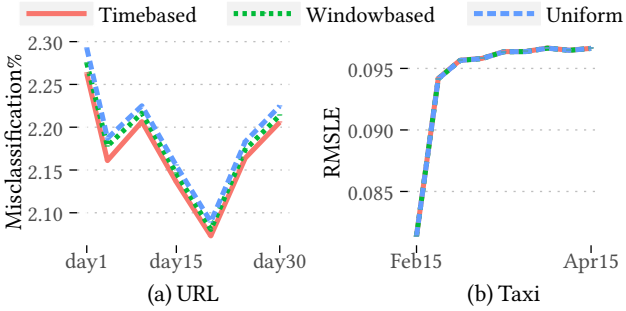


Figure 5: Effect of different sampling methods on quality

5.4 Experiment 3: Optimizations Effects

In this experiment, we analyze the effects of the optimizations, namely, online statistics computation and materialization of features, on the cost of the continuous deployment (total data preprocessing, and model training time). Figure 6 shows the effect of optimizations on deployment cost. Online statistics computation and feature materialization eliminate data preprocessing steps. As a result, the proactive training computes the gradient and updates the deployed model without the need for data preprocessing. Our optimizations improve the deployment cost by 70% and 130% for the URL and Taxi dataset, respectively. The Taxi pipeline preprocesses a larger amount of data than the URL pipeline. As a result, the reduction in the Taxi dataset is greater than the URL dataset.

5.5 Discussion

Trade-off between quality and training cost. In many real-world use cases, even a small improvement in the quality of the deployed model can have a significant impact [19]. Therefore, one can employ more complex pipelines and machine learning training algorithms to train better models. However, during the deployment where prediction queries and training data become available at a high rate, one must consider the effect of the training time. To ensure the model is always up-to-date, the platform must constantly update the model. Long retraining time may have a negative impact on the prediction

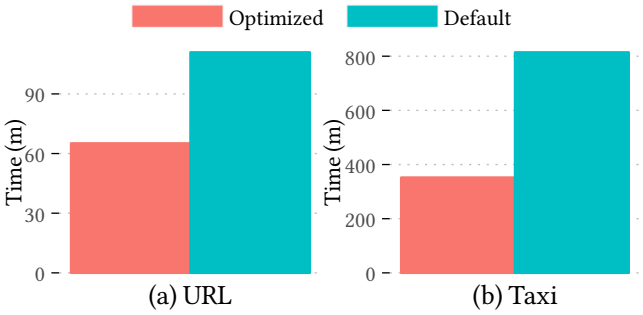


Figure 6: Effect of the optimizations on total training time

accuracy as the deployed model becomes stale. Figure 7 shows the trade-off between the average quality and the total cost of the deployment. By utilizing continuous deployment, we improve the average quality by 0.05% and 0.3% for the Taxi and URL datasets over the periodical deployment approach. We also reduce the cost of the deployment 5 to 13 times when compared with periodical deployment.

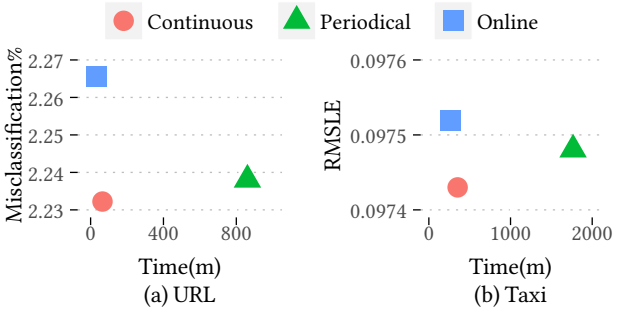


Figure 7: Trade-off between average quality and deployment cost

Staleness of the model during the periodical deployment. In the experiments of the periodical deployment approach, we pause the inflow of the training data and prediction queries. However, in real-world scenarios, the training data and the prediction queries constantly arrive at the platform. Therefore, the periodical deployment platform pauses the online update of the deployed model and answers the prediction queries using the currently deployed model (similar to how Velox operates [9]). As a result, the error rate of the deployed model may increase during the retraining process. However, in our continuous deployment platform, the average time for the proactive training is small (200 ms for the URL dataset and 700 ms for the Taxi dataset). Therefore, the continuous deployment platform always performs online model update and answers the predictions queries using an up-to-date model.

6 RELATED WORK

Traditional machine learning systems focus solely on training models and leave the task of deploying and maintaining the models to the users. It has only been recently that some platforms, for example LongView [1], Velox [9], Clipper [10] , and TensorFlow Extended [3] have proposed architectures that also consider model deployment and query answering.

LongView integrates predictive machine learning models into relational databases. It answers predictive queries and maintains and manages the models. LongView uses techniques such as query optimization and materialized view selection to increase the performance of the system. However, it only works with batch data and does not provide support for real-time queries. As a result, it does not support continuous and online learning. In contrast, our platform is designed to work in a dynamic environment where it answers prediction queries in real-time and continuously updates the model.

Velox is an implementation of the common periodical deployment approach. Velox supports online learning and can answer prediction queries in real-time. It also eliminates the need for the users to manually retrain the model offline. Velox monitors the error rate of the model using a validation set. Once the error rate exceeds a predefined threshold, Velox initiates a retraining of the model using Apache Spark. However, Velox has four drawbacks. First, retraining discards the updates that have been applied to the model so far. Second, the process of retraining on the full dataset is resource intensive and time-consuming. Third, the platform must disable online learning during the retraining. Lastly, the platform only deploys the final model and does not support the deployment of the machine learning pipeline. Our approach differs from Velox as it exploits the underlying properties of SGD to integrate the training process into the platform's workflow. Our platform replaces the offline retraining with proactive training. As a result, our deployment platform maintains the model quality with a small training cost. Moreover, our deployment platform deploys the machine learning pipeline alongside the model.

Clipper is another machine learning deployment platform that focuses on producing high quality predictions by maintaining an ensemble of models. For every prediction query, Clipper examines the confidence of every deployed model. Then, it selects the deployed model with the highest confidence for answering the prediction query. However, it does not update the deployed models, which over time leads to outdated models. On the other hand, our deployment platform focuses on maintenance and continuous update of the deployed models.

TensorFlow Extended (TFX) is a platform that supports the deployment of machine learning pipelines and models. TFX automatically stores new training data, performs analysis and validation of the data, retrain new models, and finally redeploy the new pipelines and models. Moreover, TFX supports the warm starting optimization to speed up the process of training new models. TFX aims to simplify the process of design and training of machine learning pipelines and models, simplify the platform configuration, provide platform stability, and minimize the disruptions in the deployment platform. For use cases that require months to deploy new models, TFX reduces the time to production from the order of months to weeks. Although TFX uses the term "continuous training" to describe the deployment platform, it still periodically retrains the deployed model on the historical dataset. On the contrary, our continuous deployment platform performs more rapid updates to the deployed model. By exploiting the properties of SGD optimization technique, our deployment platform rapidly updates the deployed models (seconds to minutes instead of several days or weeks) without increasing the overhead. Our proactive training component can be integrated into the TFX platform to speed up the process of pipeline and model update.

Weka [14], Apache Mahout [23], and Madlib [15] are systems that provide the necessary toolkits to train machine learning models. All of these systems provide a range of training algorithms for machine

learning methods. However, they do not support the management and deployment of machine learning models and pipelines. Our platform focuses on continuous deployment and management of machine learning pipelines and models after the initial training.

MLBase [18] and TuPaq [28] are model management systems. They provide a range of training algorithms to create machine learning models and mechanism for model search as well as model management. They focus on training high-quality models by performing automatic feature engineering and hyper-parameter search. However, they only work with batch datasets. Moreover, the users have to manually deploy the models and make them available for answering prediction queries. On the contrary, our deployment platform focuses on the continuous deployment of pipelines and models.

7 CONCLUSIONS

We propose a deployment platform for continuously updating machine learning pipelines and models. After a machine learning pipeline is designed and initially trained on a dataset, our platform deploys the pipeline and makes it available for answering prediction queries.

To guarantee a model with an acceptable error rate, existing deployment platforms periodically retrain the deployed model. However, periodical retraining is a time-consuming and resource-intensive process. As a result of the lengthy training process, the platform cannot produce fresh models. This results in model-staleness which may decrease the quality of the deployed model.

We propose a training approach, called proactive training, that utilizes samples of the historical data to train the deployed pipeline. Proactive training replaces the periodical retraining, thus guaranteeing a high-quality model without the lengthy retraining process. We also propose online statistics computation and materialization of the preprocessed features which further decreases the training time. We propose a modular design that enables our deployment platform to be integrated with different scalable data processing platforms.

We implement a prototype using Apache Spark to evaluate the performance of our deployment platform. In our experiments, we develop two pipelines with two machine learning models to process two real-world datasets. We discuss how to tune the deployment platform based on the available historical data. Our experiments show that our continuous deployment reduces the total training cost by a factor of 5 and 13 for the Taxi and URL datasets, respectively. Moreover, continuous deployment platform provides the same level of quality for the deployed model when compared with the periodical deployment approach.

In the future work, we will integrate more complex machine learning pipelines and models (e.g., neural networks) into our deployment platform and investigate the effect of concept drift and anomaly on our deployment platform.

REFERENCES

[1] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. The case for predictive database systems: Opportunities and challenges. In *CIDR*, pages 167–174, 2011.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[3] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[6] Leon Bottou, Yoshua Bengio, et al. Convergence properties of the k-means algorithms. *Advances in neural information processing systems*, pages 585–592, 1995.

[7] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[8] Olivier Chapelle. Nyc taxi & lomousine commision trip record data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. [Online; accessed 10-April-2018].

[9] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[10] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. *arXiv preprint arXiv:1612.03079*, 2016.

[11] A Philip Dawid. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, pages 278–292, 1984.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[13] Simon Funk. Netflix update: Try this at home, 2006.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[15] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[18] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[19] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 689–698. International World Wide Web Conferences Steering Committee, 2017.

[20] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying suspicious urls: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 681–688. ACM, 2009.

[21] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[23] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[24] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[25] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *ICML (3)*, 28:343–351, 2013.

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[27] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[28] Evan R Sparks, Ameet Talwalkar, Michael J Franklin, Michael I Jordan, and Tim Kraska. Tupaq: An efficient planner for large-scale predictive analytic queries. *arXiv preprint arXiv:1502.00068*, 2015.

[29] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[31] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[32] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.