Continuous Deployment of Machine Learning Pipelines

Behrouz Derakhshan
DFKI, Germany
behrouz.derakhshan@dfki.de

Alireza Rezaei Mahdiraji
DFKI, Germany
alireza.rezaei_mahdiraji@dfki.de

Tilmann Rabl*
DFKI, Germany
tilmann.rabl@dfki.de

Volker Markl†
DFKI, Germany
volker.markl@dfki.de

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 sampling techniques to include the historical data in the training process, thus eliminating the need for retraining the deployed model. We also offer online statistics computation and dynamic 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 continuous deployment reduces the total training cost up to 15 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 [22, 23]. 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 [21]. 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.

Although periodical deployment is robust and easy to tune, it cannot maintain the quality of the deployed model without incurring a high training cost. We propose a deployment platform that 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 robust, i.e., it accommodates many different types of machine learning models and pipelines. Moreover, similar to the periodical deployment, the tuning process of our deployment platform is simple and requires the same amount of user interaction as the periodical deployment.

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 the 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 features:

Proactive training. Proactive training is the process of utilizing samples of the 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 one order of magnitude while providing the same level of quality when compared to the periodical deployment approach.

Online Statistics Computation and Dynamic 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

*Also with TU Berlin, Germany, rabl@tu-berlin.de.
†Also with TU Berlin, Germany, volker.markl@tu-berlin.de.

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. In presence of a limited storage capacity, the platform removes the older transformed features, and only re-materializes them when needed (through a process called *dynamic materialization*). 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 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 dynamic 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 provide background information on the optimization strategy we utilize in our continuous deployment platform and tuning mechanism of the existing deployment platforms. 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 compute partial updates based on the current model parameters and a combination of the incoming and existing data. To compute the partial updates, we utilize Stochastic Gradient Descent (SGD) [35]. SGD has several parameters (typically referred to as hyperparameters) and in order to work effectively, they have to be tuned. In this section, we describe the details of SGD and its hyperparameters and discuss the effect of the hyperparameters 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 [23, 35], clustering [6], and matrix factorization [14, 19] utilize SGD in training models. SGD is also the most common optimization strategy for training neural networks on large datasets [13]. 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 [29] and mastering Atari games [24].

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:

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

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:

$$w^{t+1} = w^t + \eta \sum_{i \in S} x^i (y^i - \hat{P}(Y^i = 1|x^i w)) \tag{2}$$

where η is the learning rate hyperparameter 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 hyperparameter 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 [27]. Adaptive learning rate methods such as Momentum [26], Adam [18], Rmsprop [32], and AdaDelta [34] 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 [18, 32, 34]. 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 hyperparameter of stochastic gradient descent is the sample size (sometimes referred to as the mini-batch 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.

2.2 Tuning the Periodical Deployment

Typically, two groups of hyperparameters affect the efficiency of the periodical deployment approach. The first group (the deployment hyperparameters) control the frequency and amount of data for every retraining. The second group (the training hyperparameters) tune the algorithm for retraining procedure. In this work, we are targeting training algorithms based on Stochastic gradient descent. Therefore, the hyperparameters are the learning rate and the sample size.

There are several existing approaches for tuning the training hyperparameters, such as grid search, random search, and sequential model based search [4]. The deployment hyperparameters, however, are typically selected to fit the specific use case. For example, in many of the real-world use cases, one retrains the deployed model using the entire historical data on a daily basis. Similarly, when tuning our continuous deployment platform, one has to select two hyperparameters which vary from use-case to use-case. In the next sections, we show that tuning our continuous deployment approach is no more complex than tuning the periodical deployment approach.

3 CONTINUOUS TRAINING APPROACH

In this section, we describe the details of our continuous training approach. Figure 1 shows the workflow of our proposed platform. The platform processes the incoming training data through 5 stages:

1. Discretizing the data: To efficiently preprocess the data and update the model, the platform transforms the data into small chunks and stores them in the storage unit. The platform assigns a timestamp to every chunk indicating its creation time. The timestamp acts as both a unique identifier and an indicator of the recency of the chunk.

2. Preprocessing the data: The platform utilizes the deployed pipeline to preprocess the raw training data chunks and transform them into feature chunks. Then, the platform stores the feature chunks along with a reference to the originating raw data chunk in the storage unit. During the preprocessing stage, we utilize *online statistics computation* to compute the required statistics for the different pipeline components. These statistics speed up the data processing in later stages. When the storage unit becomes full, the platform starts removing the oldest feature chunks and only keep the reference to the originating raw data chunks. In case the later stages of the deployment platform request a deleted feature chunk, the platform can recreate the feature chunk by utilizing the referenced raw data chunk.

3. Sampling the data: A sampler unit samples the feature chunks from the storage. Different sampling strategies are available to address different use-case requirements.

4. Materializing the data: Depending on the size of the storage unit, some preprocessed feature chunks (results of step 2) are not materialized. If the sampler selects unmaterialized feature chunks, the platform recreates these feature chunks by utilizing the deployed pipeline through a process called *dynamic materialization*.

5. Updating the model: By utilizing the preprocessed feature chunks, the platform updates the deployed model through a process called *proactive training*.

After the user designs and deploys the pipeline, we rely on the existing big data processing frameworks to perform the discretization and preprocessing. The sampling strategy typically depends on the use-case. However, different sampling strategies have different implications for the dynamic materialization process. In the rest of this section, we first describe the details of the online statistics computation. Then we introduce the dynamic materialization approach and the effects of different sampling strategies on the materialization process. Finally, we describe the details of the proactive training method.

3.1 Online Statistics Computation

Some components of the machine learning pipeline, such as the standard scaler or the one-hot encoder, require some statistics of the dataset before they process the data. Computing these statistics requires 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. Every pipeline component first reads the incoming data. Then it updates its underlying statistics. Finally, the component transforms and forwards the data to the next component. Online computation of the required statistics eliminates the need to recompute the same statistics during the dynamic materialization and proactive training.

Online statistics computation is only applicable to certain types of pipeline components. The support for stateless pipeline components is trivial as they do not rely on any statistics before transforming the data. For stateful operations, since the statistics update occurs during the online data processing, the platform can only update the statistics that can be computed incrementally. Many of the well-known data preprocessing components (such as standardization and one-hot encoding) require statistics that can be computed incrementally (such as mean, standard deviation, and hash table). However, some pipeline components require statistics that cannot be updated incrementally (such as percentile). For these types of statistics, the pipeline component has to rescan the entire data points whenever new data becomes available. For such components, when applicable, we utilize probabilistic and approximate data structures (such as bloom filters or count-min sketch [9]). Otherwise, we do not provide support for the deployment of pipelines containing components that require a rescan of the entire data whenever new data becomes available.

The platform can also facilitate the online statistics computation for user-defined pipeline components. In Section 4, we describe how users can incorporate this feature into their custom pipeline components.

3.2 Dynamic Materialization

In order to update the statistics of the pipeline components, each component must first transform the data and then forwards the transformed data to the next component. At the end of this process, the pipeline has transformed the data chunks into feature chunks that the model will utilize during the training process. In our continuous deployment platform, we repeatedly sample the data chunks to update the model. Storing the chunks as materialized features greatly reduces the processing time as the entire data preprocessing steps can be skipped during the model update. However, in presence of a limited storage capacity, one has to consider the effect of storing the materialized feature chunks.

To address the storage capacity issue, we utilize dynamic materialization. While creating the feature chunks, the platform assigns a unique identifier (the creation timestamp) and a reference to the originating raw data chunk. In dynamic materialization, when the size of the stored feature chunks exceeds the storage capacity, the platform removes the content of the oldest materialized feature chunks from the storage and only keeps the unique identifier and the reference to the raw data chunk (similar to cache eviction). The

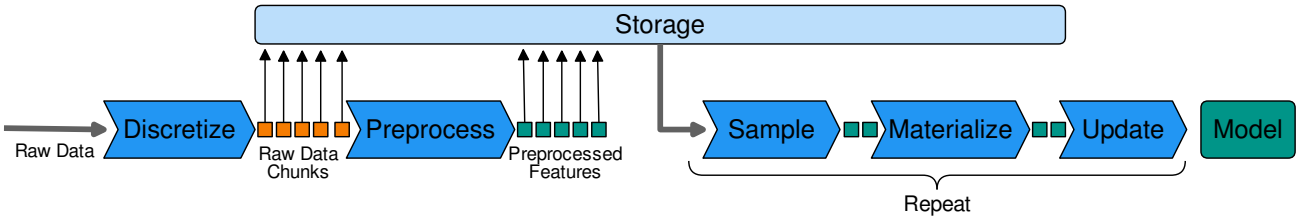

Figure 1: Workflow of our continuous deployment approach. (1) The platform converts the data into small units (2) The platform utilizes the deployed pipeline to preprocess the data and transform the raw data into features and store them in the storage. (3) The platform samples the data from the storage. (4) The platform materializes the sampled data (5) Using the sampled data, the deployment platform updates the deployed model.

next time the sampler selects one or more of the evicted feature chunks, the platform re-materializes each feature chunk from the raw data chunk by reapplying the deployed pipeline to the raw data chunk. Figure 2 shows the process of dynamic materialization in two possible scenarios. For both scenarios, there are a total 6 data chunks (raw and feature) available in the storage (with timestamps t_0 to t_5). The sampling operation selects the chunks at t_0, t_2, and t_5. In Scenario 1, all the feature chunks are materialized. Therefore, the platform directly utilizes them to update the model. In Scenario 2, the platform has previously evicted some of the materialized feature chunks due to the limit on the storage capacity. In this scenario, the platform first re-materializes the evicted chunks using the deployed pipeline components before updating the model.

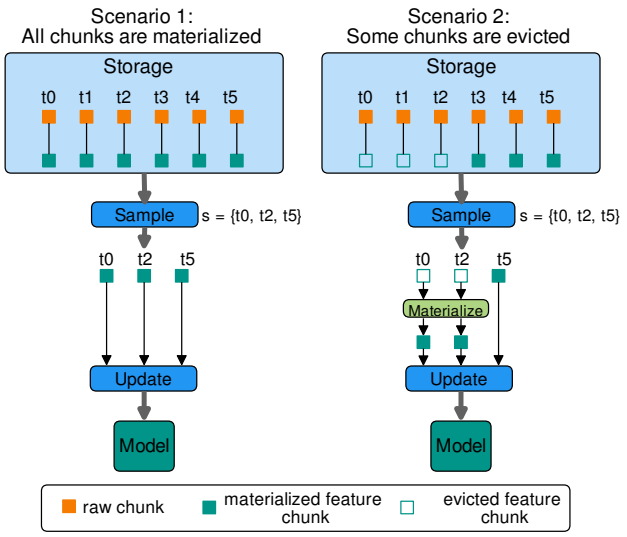

Figure 2: Dynamic Materialization process

It is important to note that the continuous training platform assumes the raw data chunks are always stored and are available for re-materialization. If some of the raw data chunks are not available, the platform ignores these chunks during the sampling operation. A similar issue arises in the periodical deployment approach. If there is not enough space to store all the historical and incoming data, at every retraining, the platform only utilizes the available data in the storage.

3.2.1 Storage requirement for materialized feature chunks. In order to estimate the storage requirement for the preprocessed feature chunks, we investigate the size complexity of different pipeline components in terms of the input size (raw data chunks). Table 1 shows the categories of the pipeline components and their characteristics. Let us assume the total number of the values in a dataset where $\mathcal{R}$ represents the rows and C represents columns is p, where $p = |\mathcal{R}| \times |C|$. Data transformation and feature selection operations either perform a one-to-one mapping (e.g., normalization) or remove some rows or columns (e.g., anomaly filtering and

Component type	Unit of work	Characteristics
data transformation	data point (row)	filtering or mapping
feature selection	feature (column)	selecting some columns
feature extraction	feature (column)	generating new columns

Table 1: Description of the pipeline component types. Unit of work indicates whether the component operates on a row or a column.

variance thresholding). Therefore, the complexity of data transformation and feature selection operations is linear in terms of the input size ($O(p)$). The case for feature extraction is more complicated as there are different types of feature extraction operations. In many cases, the feature extraction process creates a new feature (column) by combining one or more existing features (such as summing or multiplying features together). This results in a complexity of $O(p)$ as the increase in size is linear with respect to the input size. However, in some case, the feature extraction process generates many features (columns) from a small subset of the existing features. Prominent examples of such operations are one-hot encoding and feature hashing. One-hot encoding converts a column of the data with categorical values into several columns (1 column for each unique value). For every value in the original column, the encoded representation has the value of 1 in the column the value represents and 0 in all the other columns. Consider the case when we are applying the one-hot encoding operation to every column $\forall c \in C$. Furthermore, let us assume $q = \max_{\forall c \in C} |\mathcal{U}(c)|$, where $\mathcal{U}$ is the function that returns the unique values in a column ($\mathcal{U}(x) \in [1, |\mathcal{R}|]$). Thus, the complexity of the one-hot encoding operation is $O(pq)$ (each existing value is encoded with at most q binary values). Based on the value of q, two scenarios may occur:

- if $q \ll |\mathcal{R}| \Rightarrow O(pq) = O(p)$
- if $q \approx |\mathcal{R}| \Rightarrow O(pq) = O(p|\mathcal{R}|) = O(p\frac{p}{|C|}) = O(p^2)$

The second scenario represents the worst-case scenario where almost every value is unique and we have very few columns (if the number of columns is large then the complexity is lower than $O(p^2)$). A quadratic growth rate, especially in the presence of large datasets, is not desirable and may render the storage of even a few feature chunks impossible. However, both one-hot encoding and feature hashing produce sparse data where for every encoded data point, only one entry is 1 and all the other entries are 0. Therefore, by utilizing sparse vector representation, we guarantee a complexity of $O(p)$.

Since the complexity is in worst-case scenario linear with respect to the size of the input data and the eviction policy gradually dematerializes the older feature chunks, the platform ensures the size of the materialized features will not unexpectedly exceed the storage capacity.

3.2.2 Effects of sampling strategies on the dynamic materialization. Our platform offers three sampling strategies, namely, uniform, window-based, and time-based (Section 4.2). The choice of the sampling strategy affects the efficiency of the dynamic materialization. Here, we analyze the effects of dynamic materialization in reducing the data processing overhead.

We define N as the maximum number of the raw data chunks, n as the number of existing raw chunks during a sampling operation, m as the maximum number of the materialized feature chunks (corresponds to the size of the dedicated storage for the materialized feature chunks), and s as the sample size (in each sampling operation, we are sampling s chunks out of the available n chunks)[1]. Let us define ms as the number of materialized feature chunks in a sampling operation. The variable ms follows a hypergeometric distribution[2] (sampling without replacement) where the number of success states is m, and the number of draws is s. Therefore, the expected value of ms for a sampling operation with n chunks is:

$$E_n[ms] = s\frac{m}{n}$$

To quantify the efficiency of the dynamic materialization, we introduce the materialization utilization rate with n raw chunks, which indicates the ratio of the materialized feature chunks:

$$MU_n = \frac{E_n[ms]}{s}$$

Finally, the average materialization utilization rate for the dynamic materialization process is:

$$MU = \frac{\sum_{n=1}^{N} MU_n}{N} \tag{3}$$

MU indicates the ratio of the feature chunks that do not require re-materialization before updating the model (an MU of 0.5 shows on average half of the sampled chunks are materialized). To simplify the analysis, we assume the platform performs one sampling operation after every incoming data chunk. In reality, a scheduler component governs the frequency of the sampling operation (Section 4.1). Next, we describe how the sampling strategy affects the computation of MU.

Random Sampling: For the random sampling strategy, we compute MU_n as:

$$MU_n = \begin{cases} 1, & \text{if } n \le m \\ \frac{E_n[ms]}{s} = \frac{s\frac{m}{n}}{s} = \frac{m}{n}, & \text{otherwise} \end{cases}$$

Since for the first m sampling operations the number of raw chunks (n) is smaller than the total size of the materialized chunks (m), MU_n is 1.0 (every sampled chunk is materialized).

$$\begin{aligned} MU = \frac{\sum_{n=1}^{N} MU_n}{N} &= \frac{m \times 1.0 + \sum_{n=m+1}^{N} \frac{m}{n}}{N} \\ &= \frac{m + m(\frac{1}{m+1} + \frac{1}{m+2} + \ldots + \frac{1}{N})}{N} \\ &= \frac{m(1 + (H_N - H_{m+1}))}{N} \\ &\approx \frac{m(1 + ln(N) - ln(m+1))}{N} \end{aligned} \tag{4}$$

The highlighted section corresponds to the Harmonic numbers [31]. The t-th harmonic number is:

$$H_t = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{t} \approx ln(t) + \gamma + \frac{1}{2t} - \frac{1}{12t^2}$$

where $\gamma \approx 0.5772156649$ is the Euler-Mascheroni constant. In our analysis, since t is sufficiently large (more than 1000), we ignore $\frac{1}{2t} - \frac{1}{12t^2}$.

Window-based Sampling: In the window-based sampling, we have an extra parameter w which indicates the number of chunks in the active window. If $m \ge w$ then $MU = 1$, as all the feature chunks in the active window are always materialized. However, when $m < w$:

$$MU_n = \begin{cases} 1, & \text{if } n \le m \\ \frac{E_n[ms]}{s} = \frac{m}{n}, & \text{if } m < n \le w \\ \frac{E_w[ms]}{s} = \frac{m}{w}, & \text{if } w < n \end{cases}$$

therefore:

$$\begin{aligned} MU = \frac{\sum_{n=1}^{N} MU_n}{N} &= \frac{m + \sum_{n=m+1}^{w} \frac{m}{n} + (N-w)\frac{m}{w}}{N} \\ &\approx \frac{m + m(H_w - H_{m+1}) + (N-w)\frac{m}{w}}{N} \\ &= \frac{m(1 + ln(w) - ln(m+1) + \frac{N-w}{w})}{N} \end{aligned} \tag{5}$$

Time-based Sampling: For the time-based sampling strategy, there is no direct approach for computing the expected value of ms (the number of the materialized chunks in the sample). However, we are assigning a higher sampling probability to the recent chunks. As a result, we guarantee the time-based sampling has a higher average materialization utilization rate than the uniform sampling. In the experiments, we empirically show the average materialization utilization rate.

In our experiments, we execute a deployment scenario with a total of 12,000 chunks ($N = 12000$), where each chunk is around 3.5 MB (a total of 42 GB). For the uniform sampling strategy, in order to achieve $MU = 0.91$, using Formula 4, we set the maximum number of the materialized chunks to 7,200 ($m = 7200$). This shows that, in the worst-case scenario (when uniform sampling is utilized), by materializing around 25 GB of the data, we ensure the deployment platform does not need to re-materialize the data 91% of the time.

3.3 Proactive Training

Updating the model is the last step of our continuous deployment platform. We update the model through the proactive training process. Unlike, the full retraining process that is triggered by a certain event (such as a drop in the quality or certain amount of time elapsed since the last retraining), proactive training continuously updates the deployed model. The proactive training utilizes the mini-batch stochastic gradient descent to update the model incrementally. Each instance of the proactive training is analogous to an iteration of the mini-batch SGD. Algorithm 1 shows the pseudocode of the mini-batch SGD algorithm. Since the platform executes the proactive training in arbitrary intervals, we must ensure each instance of the proactive training is independent of the previous instances. According to the mini-batch SGD algorithm, each iteration of the SGD only requires the model (m_{i-1}) and the learning rate (η_{i-1}) of the previous iteration (Lines 4 and 5). Given these parameters, iterations of SGD are conditionally independent

Algorithm 1 mini-batch Stochastic Gradient Descent

Input: D = training dataset
Output: m = trained model
1: initialize m_0
2: **for** $i = 1 \ldots n$ **do**
3: s_i = sample from D
4: $g = \nabla J(s_i, m_{i-1})$
5: $m_i = m_{i-1} - \eta_{i-1} g$
6: **end for**
7: **return** m_n

[1] The value N corresponds to the size of the storage unit dedicated for raw data chunks which bounds the variable n. If we assume n is unbounded, then as $lim_{n \to \infty}$, the probability of sampling materialized feature chunks becomes 0.
[2] https://en.wikipedia.org/wiki/Hypergeometric_distribution

of each other. Therefore, to execute the proactive training, the deployment platform only needs to store the model weights and the learning rate. By proactively training the deployed model, the platform ensures the model stays up-to-date and provides accurate predictions.

Proactive training is a form of incremental training [15] which is limited to SGD-based models. In our deployment platform, one can replace the proactive training with other forms of incremental training. However, we limit the platform's support to SGD for two reasons. First, SGD is simple to implement and is used for training a verity of machine learning models in different domains [6, 14, 19, 23]. Second, since the combination of the data sampling and the proactive training is similar to the mini-batch SGD procedure, proactive training provides the same regret bound on the convergence rate as the existing stochastic optimization approaches [18, 35].

4 DEPLOYMENT PLATFORM

Our proposed deployment platform comprises of five main components: pipeline manager, data manager, scheduler, proactive trainer, and execution engine. Figure 3 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 predictions, 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:

$$T' = S * T * pr * pl \tag{6}$$

where T' is the time in seconds when the next proactive training is scheduled to execute, T is the length of the execution time (in seconds) of the previous proactive training, pl is the average prediction latency (second per item), and pr is the average number of prediction queries per second (items 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 ($1 \leq S \leq 2$) results in smaller scheduling intervals. As a result, the deployment platform allocates more resources for training the model.

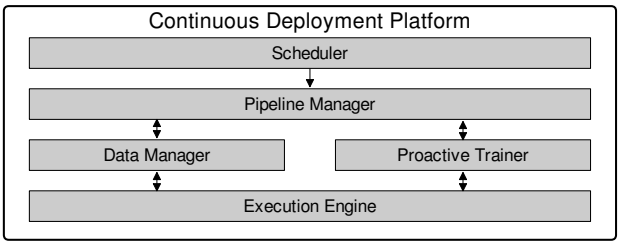

Figure 3: Architecture of the Continuous Deployment Platform

4.2 Data Manager

The data manager component is responsible for the storage of historical data and materialized features, receiving the incoming training data, and providing the pipeline manager with samples of the data. The data manager has four main tasks. First, the data manager discretizes the incoming training data into chunks, assigns a timestamp (which acts as a unique identifier) to them, and stores them in the storage unit. Second, it forwards the data chunks (and the prediction queries) to the pipeline manager for further processing. Third, after the pipeline manager transforms the data into feature chunks, the data manager stores the transformed feature chunks in the storage unit along with a reference to the originating raw data chunk (i.e., the timestamp of the raw data chunk). If the storage unit reaches its limit, the data manager removes old feature chunks. Finally, upon the request of the pipeline manager, the data manager samples the data for proactive training.

During the sampling procedure, the data manager randomly selects a set of chunks by using their timestamp as key. Then, the data manager proceeds as follows. For every sampled timestamp, if the transformed feature chunk exists in the storage, then the data manager forwards it to the pipeline manager. However, if the data manager has previously removed the transformed feature chunks from the storage unit, the data manager forwards the raw data chunk to the pipeline manager and notifies the pipeline manager to re-transform the raw data chunk (i.e., the dynamic materialization process).

The data manager provides three sampling strategies, namely, uniform, time-based, and window-based. The uniform sampling strategy provides a random sample from the entire data where every data chunk has the same probability of being sampled. The time-based sampling strategy assigns weights to every data chunk based on their timestamp such that recent chunks have a higher probability of being sampled. The window-based sampling strategy 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. Based on the specific use-case, the user chooses the appropriate sampling strategy. 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 training. However, in some use cases, the incoming training data is not time-dependent (e.g., image classification of objects). In these scenarios, the window-based and the time-based sampling strategies may fail to provide a non-biased sample. In Section 5, we evaluate the effect of the sampling strategy on both the total deployment cost and the quality of the deployed model.

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.

Each pipeline component must implement two methods: *update* and *transform*. Furthermore, every pipeline component has an internal state for storing the statistics (if needed). During the online training, when new training data becomes available, the pipeline manager first invokes the *update* method which enables the component to update its internal statistics using the incoming data. Then, the pipeline manager invokes the *transform* method, which transforms the data. After forwarding the data through every component of the pipeline, the pipeline manager sends the transformed features to the data manager for storage.

When the scheduler component informs the pipeline manager to execute proactive training, the pipeline manager requests the data manager to provide it with a sample of the data chunks for the next proactive training. If some of the sampled data chunks are not materialized, the pipeline manager re-materializes the chunks by invoking the transform methods of the pipeline components. Then, it provides the proactive trainer with the current model parameters and the materialized sample of the features. 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 (by only invoking the *transform* method of every pipeline component). 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]. Finally, the pipeline manager utilizes 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 (sampled materialized features) 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.

In order for the proactive training to update the deployed model, the machine learning model component of the deployed pipeline must implement an *update* method, which is responsible for computing the gradient. To provide support for other types of incremental training approaches, one needs to implement the training logic in the *update* method of the model. However, as described in Section 3.3, the proactive training with data sampling can guarantee convergence only when the SGD optimization is utilized.

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 [33], Apache Flink [7], and GoogleDataFlow [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 dynamic materialization optimizations 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 [22]. 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[3]. 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[4], the bearing[5], 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, 16 GB of RAM, and 512GB SSD 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 [28]. We leverage the SVM, LogisticRegression, and the GradientDescent classes of the machine learning library in Spark (MLlib) to implement the proactive training logic. We represent both the raw data and the feature chunks as RDDs. Therefore, we can utilize the caching mechanism of Apache Spark to simply materialize/dematerialize feature chunks.

Datasets. Table 2 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

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

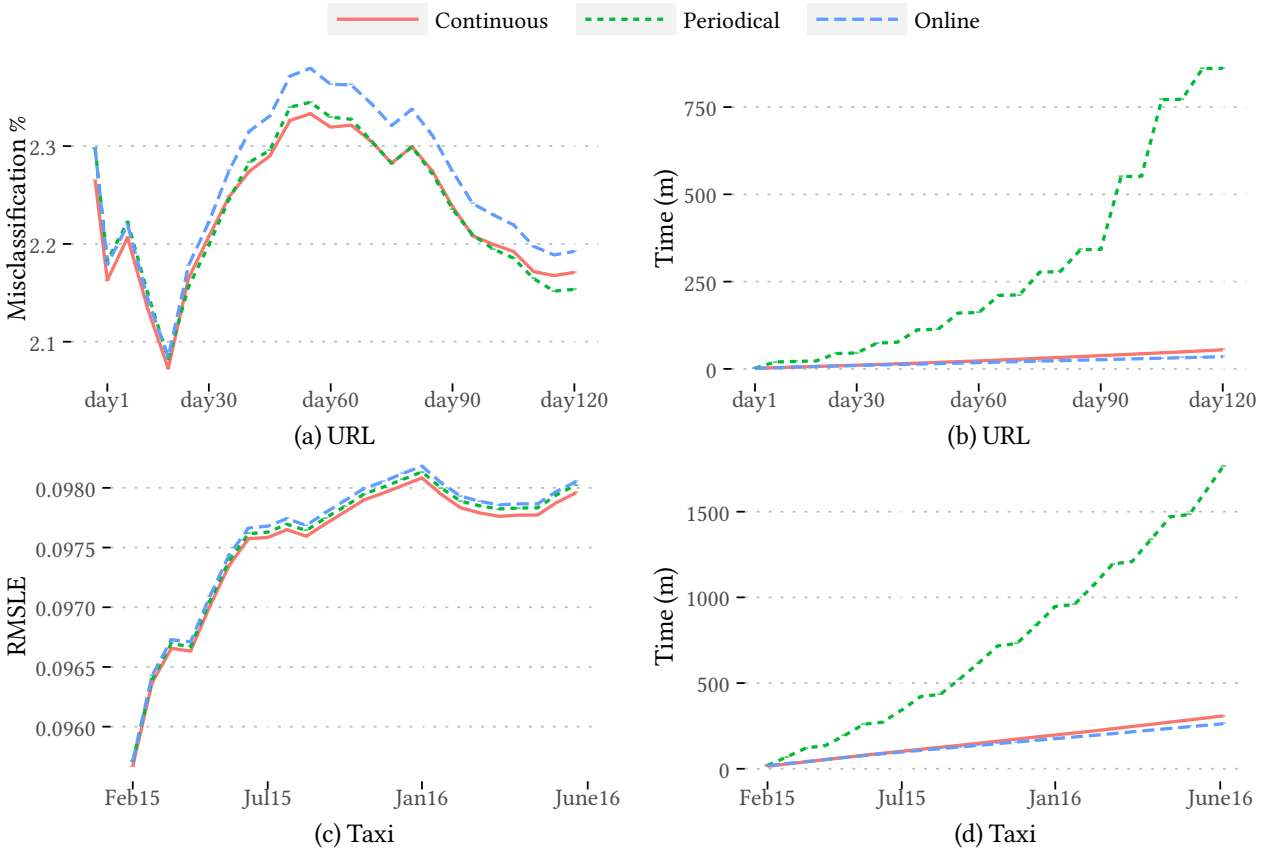

Figure 4: Model Quality and Training cost for different deployment approaches

deploy the models (and the pipelines). We use the remaining data for sending prediction queries and further training of the deployed models.

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 2: 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)

Evaluation metrics. 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 [12]. For experiments that capture the cost of the deployment, we measure the time the platforms spend in updating the model, performing proactive training (retraining for the periodical deployment scenario), and answering prediction queries.

Deployment process. The URL dataset does not have timestamps. Therefore, we divide every day of the data into chunks of 1 minute which results in a total of 12000 chunks, each one with the size of roughly 200KB. The deployment platform first uses the chunks for prequential evaluation and then updates the deployed model. The Taxi dataset includes timestamps. In our experiments, each chunk of the Taxi dataset contains one hour of the data, which results in a total of 12382 chunks, with an average size of 3MB per chunk. The deployment platform processes the chunks in order of the timestamps (from 2015-Feb-01 00:00 to 2016-Jun-30 24:00, an 18 months period).

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.

- Online: deploy the pipeline, then utilize online gradient descent with Adam learning rate adaptation 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.

Figure 4 (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 4 (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 4 (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 4 (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 4 (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 the end of the deployment is only 50% larger than the online deployment approach (around 54 minutes). Because of the online statistics computation and the dynamic 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 15 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 4 (d)), the cost of online, continuous, and periodical deployments are 262, 308, and 1765 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 6 times larger than the continuous deployment (instead of 15 times for URL dataset).

5.3 Experiment 2: System Tuning

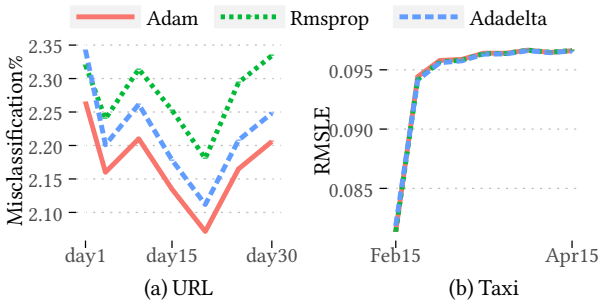

Figure 5: Result of hyperparameter tuning during the deployment

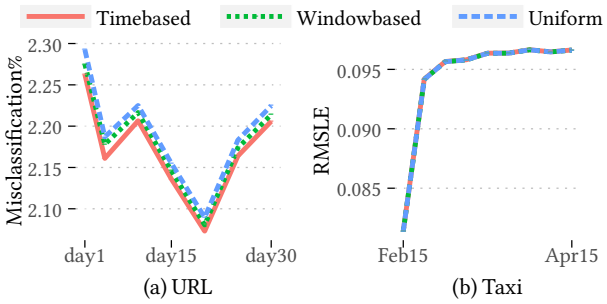

Figure 6: Effect of different sampling methods on quality

In this experiment, we investigate the effect of different parameters on the quality of the models after deployment. As described in Section 3.3, proactive training is an extension of the stochastic gradient descent to the deployment phase. Therefore, we expect the set of hyperparameters with the 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 2) 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 3 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 the 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 5 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.

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 6 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

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

and uniform sampling, respectively. As new features are added to the URL dataset over time, the underlying characteristics of the dataset gradually change [22]. 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 are 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.

5.4 Experiment 3: Optimizations Effects

In this experiment, we analyze the effect of the system optimizations, i.e., online statistics computation and the dynamic materialization on the total deployment cost. We define the materialization rate as the ratio of the number of materialized chunks over the total number of chunks (both URL and Taxi have around 12,000 chunks in total). For both datasets, the materialization rates of 0.0, 0.2, 0.6, and 1.0 indicates that 0, 2400, 7200, and 12000 chunks are materialized. For the window-based sampling strategy, we set the window size to 6,000 chunks (half of the total chunks). Using the Formula 4

	URL		Taxi	
Sampling	m=0.2	m=0.6	m=0.2	m=0.6
Uniform	0.52	0.91	0.51	0.90
Window-based	0.58	1.0	0.57	1.0
Time-based	0.68	0.97	0.65	0.97

Table 4: Empirical computation of *MU* for different sampling strategies and materialization rates (m).

and 5 of Section 3.2, when the materialization rate is 0.0, 0.2, 0.6, and 1.0 the average materialization utilization rate (*MU*) is 0.0, 0.5218, 0.9064, and 1.0 for uniform sampling and 0.0, 0.5832, 1.0, and 1.0 for window-based sampling. To validate our analysis, we compute *MU* empirically as well. Table 4 shows the value of *MU* for different settings. For both the uniform and time-based sampling, the empirical and analytical computation yield similar values. Moreover, the empirical computation shows that the time-based strategy performs better than the uniform sampling strategy. When the number of materialized feature chunks is 0 or 12000, the design of the deployment platform guarantees that *MU* is 0.0 and 1.0, respectively. Therefore, we do not report those results in the table.

To examine the effect of *MU* on the deployment cost, we plot the total deployment cost using different sampling strategies and materialization rates for the URL and Taxi deployment scenarios in Figure 7. When the materialization rate is 0.0 or 1.0, the sampling strategies have similar effects on the deployment cost. Therefore, the total deployment cost for every sampling strategy is 90 minutes for URL and 600 minutes for Taxi deployment scenario, when the materialization rate is 0.0. Similarly, the deployment cost is 54 minutes for URL and 308 minutes for Taxi, when the materialization rate is 1.0 (an improvement of 40% for URL and 49% for Taxi deployment scenarios).

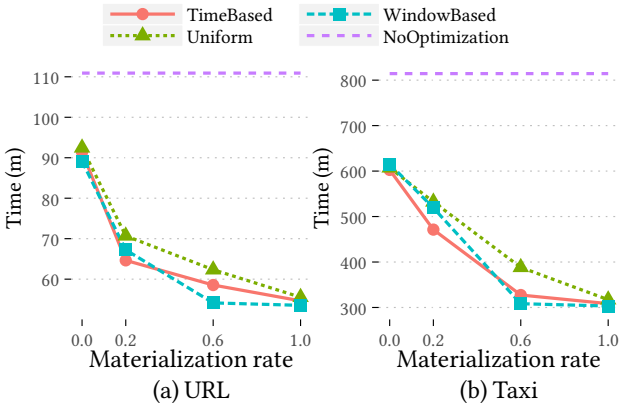

Figure 7: Effect of the online statistics computation and dynamic materialization on the deployment cost

For the URL deployment scenario, when the materialization rate is 0.2, time-based, window-based, and uniform sampling improve the deployment cost by 30%, 25%, and 23% in comparison with the materialization rate of 0.0. Similarly, in the Taxi deployment scenario, time-based, window-based, and uniform sampling improve the deployment cost by 22%, 16%, and 12%, respectively. Time-based sampling performs better since it has a higher *MU* value than the other two sampling strategies (Table 4). When the materialization rate is 0.2, the rate of the decrease in the deployment cost for the URL scenario is greater than the Taxi scenario. We attribute this difference in the decrease in the deployment cost to two reasons. First, the number of sampled chunks in the Taxi deployment scenario is larger than the URL (720 for Taxi and 100 for URL). Before updating the model, we utilize the *context.union* operation of Spark, to combine all the non-materialized and materialized chunks. The union operation incurs a larger overhead when the number of underlying chunks is bigger. Second, we execute the URL deployment scenario on a single machine with SSD. Since materializing data that resides on an SSD is faster than an HD, we observe a larger decrease in the deployment cost.

When the materialization rate is 0.6, window-based sampling has the best performance. Since the size of the window is smaller than the number of the materialized feature chunks, every sampled feature chunk is materialized. For the URL deployment scenario, window-based, time-based, and uniform sampling improves the performance by 40%, 36%, and 33%, respectively. For the Taxi deployment scenario, window-based, time-based, and uniform sampling improves the performance by 49%, 46%, and 37%, respectively. Similar phenomena explain the difference in performance improvement at materialization rate of 0.6 between the Taxi and the URL deployment scenarios. At materialization rate of 0.6, more than 90% of the chunks are materialized. Therefore, the Taxi deployment scenario gains relatively more than the URL deployment scenario from a smaller number of disk I/O operations.

To analyze the effect of the online statistics computation on the deployment cost, we also execute the deployment scenarios without the online statistics computation and the dynamic materialization optimizations. In this case, the deployment platform first accesses the sampled raw data chunk directly from the disk. Then, the platform recomputes the required statistics of every component by scanning the data. Finally, it transforms the raw data chunk

into the preprocessed feature chunks by utilizing the deployed pipeline. Without the optimizations, the choice of the sampling strategy does not affect the total deployment time (similar to the materialization rate of 0.0). Therefore, when the optimizations are disabled, we only show the results for the time-based sampling (depicted as NoOptimization in Figure 7). The extra disk access and data processing result in an increase of %110 for the URL (Figure 7a) and %170 for the Taxi deployment scenarios (Figure 7b) when compared with a fully optimized execution (with online statistics computation and materialization rate of 1.0). Similar to the dynamic materialization case, we observe a larger increase in the deployment cost of the Taxi deployment scenario due to the larger overhead of disk I/O.

The result of this experiment shows that even under limited storage we can benefit from the dynamic materialization, especially for the time-based and window-based sampling strategies. Furthermore, online statistics computation can improve the total deployment cost, especially when the expected amount of incoming data is large.

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 [21]. 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 accuracy as the deployed model becomes stale. Figure 8 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 6 to 15 times when compared with periodical deployment.

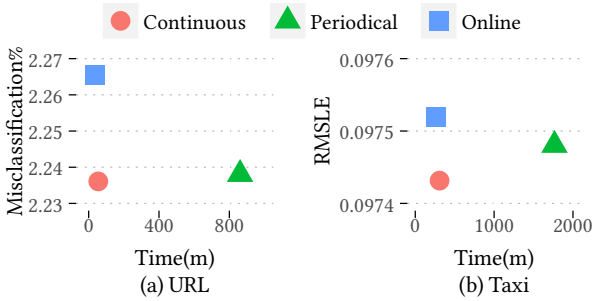

Figure 8: Trade-off between average quality and deployment cost

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 [10]). 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 the 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 [10], Clipper [11] , 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, retrains new models, and finally redeploys 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 [16], Apache Mahout [25], and Madlib [17] 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 [20] and TuPaq [30] 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 dynamic 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 deployment cost by a factor of 6 and 15 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 anomalies 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, 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, 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, et al. 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] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, et al. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[11] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, et al. Clipper: A low-latency online prediction serving system. *arXiv preprint arXiv:1612.03079*, 2016.

[12] 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.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

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

[15] Alexander Gepperth and Barbara Hammer. Incremental learning algorithms and applications. In *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[17] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

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

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

[20] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, et al. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[21] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, et al. 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.

[22] 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.

[23] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, et al. 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.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

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

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

[28] 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.

[29] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[30] 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.

[31] Zhi-Wei Sun. Arithmetic theory of harmonic numbers. *Proceedings of the American Mathematical Society*, 140(2):415–428, 2012.

[32] 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.

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

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

[35] 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.