Materialization and Reuse Optimizations for Machine Learning Workloads

vorgelegt von
M. Sc.
Behrouz Derakhshan

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Berlin 2022

Acknowledgements

First and foremost, I would like to thank my advisors Volker Markl, Tilmann Rabl, Zoi Kaoudi, and Alireza Rezaei Mahdiraji for their support and mentorship during my PhD studies. I also like to thank my PhD committee, Volker Markl, Tilmann Rabl, Carsten Binnig, and Klaus-Robert Müller for their guidance and valuable feedback.

Many fellow researchers in the Database and Information System Group of TU Berlin and Intelligent Analytics for Massive Data at DFKI deserve special recognition for their intellectual and moral support. Special thanks go to my colleagues: Alireza, Andreas, Ankit, Ariane, Asterios, Azadeh, Clemens, Claudia, Felix, Gábor, Harry, Jonas, Juan, Melanie, Mahdi, Makis, Martin, Mohammad, Sebastian, Sergey, Ventura, Viktor, Ziawasch, and Zoi.

I would like to thank my parents Khosrow and Fatemeh, my brother Behnam, my friends Armin, Farshad, Hessam, and Emad for encouraging me to pursue a doctoral degree.

Finally, I would like to thank my wonderful wife Vivi for encouraging me, guiding me, proofreading my papers and thesis, and supporting me throughout this time. This would not have been possible without you.

Abstract

Machine learning (ML) is becoming increasingly ubiquitous in industrial and scientific applications. The life cycle of ML applications consists of analyzing the input data, preprocessing and feature engineering, model selection and hyperparameter tuning, and deploying the models and pipelines for inference. Designing ML pipelines is an iterative process. Data scientists experiment with different preprocessing and feature engineering operations, ML models and hyperparameters, and evaluate the performance of each configuration. Furthermore, the deployed pipelines are continuously improved by retraining when new data becomes available. The iterative and continuous nature of ML workloads results in redundant and overlapping data processing and model training operations.

This thesis identifies three important settings where redundant and overlapping computations result in significantly higher execution costs and waste of resources. The workloads in each setting exhibit similar types of computational redundancies. We propose a suite of techniques to optimize the execution of the workloads in each setting via materialization and reuse of the redundant and overlapping computations. Our solution is inspired by database optimization techniques, such as materialized view selection and multi-query optimization.

The first setting we investigate is ML workloads in collaborative environments. In a collaborative setting, multiple data scientists collaborate and build on each other's work to incrementally improve the design of ML pipelines. In our solution, we first estimate the materialization benefit of the generated artifacts (i.e., preprocessed data, extracted features, and trained models). Then, we materialize the artifacts with the highest benefits under a storage constraint. We base the materialization benefit on several criteria, such as computation time, size, and performance of the ML models.

The second setting involves continuous training of a deployed ML pipeline. After the design and hyperparameter tuning of a pipeline, data scientists deploy the pipeline for answering prediction queries. The most common approach to maintain the model accuracy is to retrain the model on the combination of the historical data and newly available data. Our solution materializes the generated features of the pipeline. Furthermore, we introduce *Proactive Training*, which replaces the full retraining of the pipeline. In proactive training, we update the deployed model in-place by continuously applying iterations of mini-batch stochastic gradient descent optimization.

The third identified setting is the multi-pipeline deployment and continuous training. In this setting, data scientists deploy *multiple* pipelines into a production environment and retrain them following a schedule. This is a common scenario for performing A/B testing or ensemble learning. After an execution, every pipeline operator generates some artifacts, i.e., computed statistics, transformed data, or ML models. Our solution employs a materialization algorithm that given a storage budget, materializes the subset of the artifacts, which minimizes the run time of the subsequent executions. Furthermore, we propose a reuse algorithm that generates an optimal execution plan. The reuse algorithm operates on three principles. First, it shares the computation of the similar operators of the ML pipelines. Second, it reuses the materialized artifacts when appropriate. Third, it recognizes a distribution shift in data and adjusts the execution plan accordingly.

The outcome of the thesis is a set of techniques to efficiently execute ML workloads by leveraging the materialization and reuse of the artifacts (i.e., features, computed statistics, and ML models) in different settings. Our proposed optimization techniques can improve the execution time by up to one order of magnitude in different settings.

Zusammenfassung

Maschinelles Lernen (ML) wird in industriellen und wissenschaftlichen Anwendungen zunehmend präsenter. Der Lebenszyklus von ML-Anwendungen umfasst die Analyse der zugeführten Rohdaten und deren Vorverarbeitung, Feature-Engineering, die Modellauswahl und die Abstimmung der Hyperparameter sowie den Einsatz der Modelle und Pipelines für die Inferenz. Die Entwicklung von ML-Pipelines ist ein iterativer Prozess. Datenwissenschaftler*innen experimentieren mit verschiedenen Konfigurationen, wie z.B. Vorverarbeitungsschritte, Feature-Engineering-Vorgänge, ML-Modelle und Hyperparameter, und bewerten anschließend deren Leistung. Bereits eingesetzte Pipelines werden bei Verfügbarkeit von neuen Daten neu trainiert und können damit kontinuierlich verbessert werden. Die iterative Entwicklung sowie die kontinuierliche Weiterentwicklung von ML-Workloads führt zu redundanten und sich überschneidenden Datenverarbeitungs- und Modelltrainingsoperationen.

In dieser Arbeit werden drei Szenarien identifiziert, in denen redundante und überlappende Berechnungen zu deutlich höheren Ausführungskosten sowie Ressourcenverschwendung führen. Die Arbeitslasten in allen drei Szenarien weisen ähnliche Arten von Rechenredundanzen auf. Unsere Lösung nutzt eine Reihe von Techniken um die Ausführung der Arbeitslasten in jedem Szenario durch Materialisierung und Wiederverwendung der redundanten und überlappenden Berechnungen zu optimieren. Die genutzten Techniken sind inspiriert von Methoden zur Datenbankoptimierung, wie beispielsweise der Auswahl materialisierter Ansichten und der Optimierung mehrerer Abfragen.

Das erste von uns untersuchte Szenario sind ML-Workloads in kollaborativen Umgebungen. In einer kollaborativen Umgebung arbeiten mehrere Datenwissenschaftler*innen zusammen und bauen ihre Arbeit aufeinander auf, um das Design von ML-Pipelines schrittweise zu verbessern. In unserer Lösung schätzen wir zunächst den Nutzen der Materialisierung einzelner generierten Artefakte (d.h. vorverarbeitete Daten, extrahierte Merkmale und trainierte Modelle). Anschließend materialisieren wir die Artefakte mit dem höchsten Nutzen unter Berücksichtigung einer Speicherbeschränkung. Der Nutzen der Materialisierung basiert auf mehreren Kriterien, wie z. B. Rechenzeit, Größe und Leistung der ML-Modelle.

Das zweite Szenario umfasst das kontinuierliche Training einer eingesetzten ML-Pipeline. Nach dem Entwurf und der Abstimmung der Hyperparameter einer Pipeline setzen Datenwissenschaftler*innen die Pipeline zur Beantwortung von Vorhersageanfragen ein. Der gängigste Ansatz zur Aufrechterhaltung der Modellgenauigkeit besteht darin, das Modell anhand einer Kombination aus historischen und neuen Daten neu zu trainieren. Unsere Lösung materialisiert die generierten Merkmale der Pipeline. Darüber hinaus führen wir Proactive Training ein. Proactive Training nutzt die kontinuierlichen Iterationen der stochastischen

Mini-Batch-Gradientenabstiegsoptimierung, um das eingesetzte Modell an Ort und Stelle zu aktualisieren. Das dritte identifizierte Szenario ist der Einsatz mehrerer Pipelines und kontinuierliches Training. In diesem Fall setzen Datenwissenschaftler*innen mehrere Pipelines in einer Produktionsumgebung ein und trainieren diese nach einem bestimmten Zeitplan neu. Dies ist ein gängiges Szenario für die Durchführung von A/B-Tests oder Ensemble-Learning. Nach einer Ausführung erzeugt jeder Pipeline-Operator einige Artefakte, d. h. berechnete Statistiken, transformierte Daten oder ML-Modelle. Unsere Lösung verwendet einen Materialisierungsalgorithmus, der eine Teilmenge der erzeugten Artefakte materialisiert. Dabei wählt der Algorithmus Artefakte aus, welche die Laufzeit der nachfolgenden Ausführungen minimieren, und berücksichtigt ein gegebenes Speicherbudget. Außerdem enthält unsere Lösung einen Wiederverwendungsalgorithmus, der nach drei Prinzipien arbeitet. Erstens kombiniert er die Berechnung der ähnlichen Operatoren der ML-Pipelines. Zweitens werden die materialisierten Artefakte wiederverwendet, wenn dies sinnvoll ist. Drittens erkennt er eine Verteilungsverschiebung in den Daten und passt den Ausführungsplan entsprechend an. Dadurch generiert der Wiederverwendungsalgorithmus einen optimalen Ausführungsplan.

Das Ergebnis dieser Arbeit ist eine Reihe von Techniken zur effizienten Ausführung von ML-Workloads durch Nutzung von Materialisierungsverfahren und Wiederverwendung von erzeugten Artefakte (d.h. Merkmale, berechnete Statistiken und ML-Modelle) in verschiedenen Szenarien. Die von uns vorgeschlagenen Optimierungstechniken können die Ausführungszeit in den drei verschiedenen Szenarien um bis zu einer Größenordnung verbessern.

Table of Contents

List of Figures

List of Tables

1 Introduction

Machine Learning (ML) is increasingly utilized in industrial and scientific applications to gain insight from data. ML solutions are typically pipelines of several steps, from initial data analysis, data preprocessing, and feature engineering to model selection and hyperparameter tuning. Data scientists and ML practitioners typically try out many different pipeline components to find an acceptable solution. Thus, designing ML pipelines is iterative, where in each iteration, one or a set of pipeline components may change. Moreover, many ML applications have a dynamic nature, i.e., new data becomes available. Therefore, after finding an acceptable solution, one has to continuously process the data and update the ML models by utilizing the newly available data.

The iterative and continuous nature of ML workloads results in redundant and overlapping data processing and model training operations that hurt the performance of ML applications. *Inspired by existing database optimization techniques, the goal of this thesis is to identify computational redundancies in ML applications and optimize their execution.*

In database systems, similar problems exist. View materialization and reuse [1, 2, 3] and multi-query optimization [4] are three-decade-old database techniques to tackle data processing redundancies. However, existing approaches focus on the optimization of SQL queries in relational databases. Furthermore, these techniques assume the output of the operators are tuples. In ML pipelines, the output of the operators is typically generated features, computed statistics over some data, or complex ML models. Therefore, existing database optimization techniques are not directly applicable to complex ML workloads.

In ML, such ideas have also been applied. However, existing solutions either only apply to parts of the pipeline or are limited to single-session applications. Two examples of such solutions are Columbus [5] and Helix [6]. Columbus is a system that focuses on optimizing the execution of feature selection workloads by materialization and reuse of intermediates. Helix is a system that materializes and reuses intermediates within a single session. Furthermore, ML workloads increasingly utilize containerized environments for execution [7]. Containerized environments provide better resource isolation and utilization for the execution of ML workloads. However,

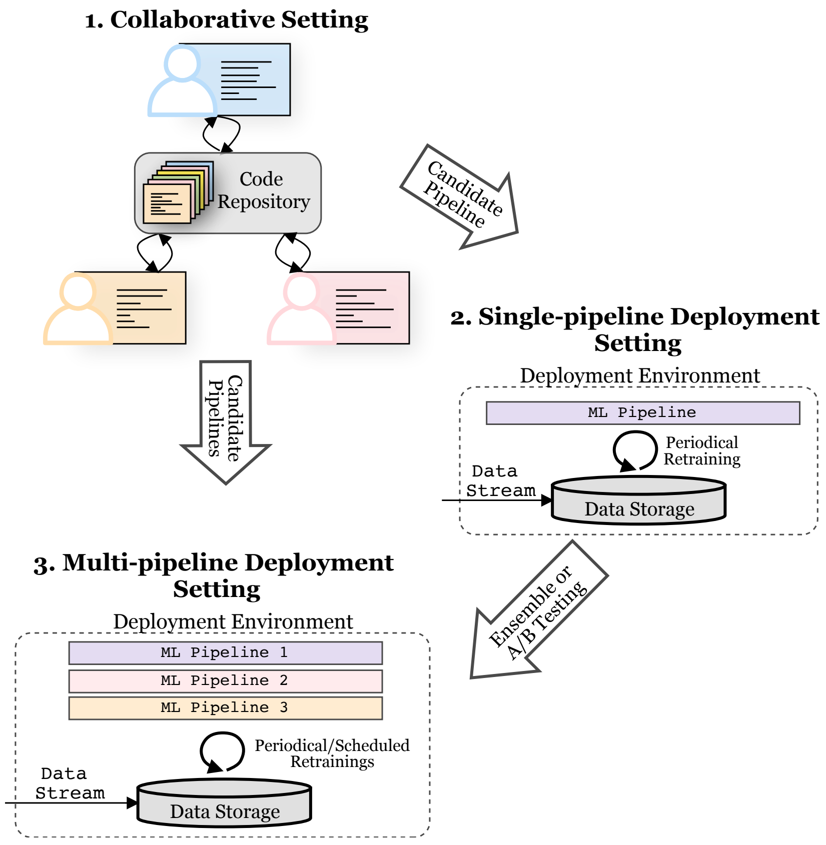


Figure 1.1: Life cycle of machine learning applications and existing workload settings.

they render the implementation of such optimizations (materialization/reuse and multi-query optimization) difficult as there is no centralized data management system.

In this thesis, we study the workloads that are part of the end-to-end ML application life cycles. We identify several settings where the workloads in each setting exhibit the same types of computational redundancies. Then, we propose different materialization and reuse optimization techniques to improve the performance of executing the workloads in each setting.

1.1 ML Workload Settings

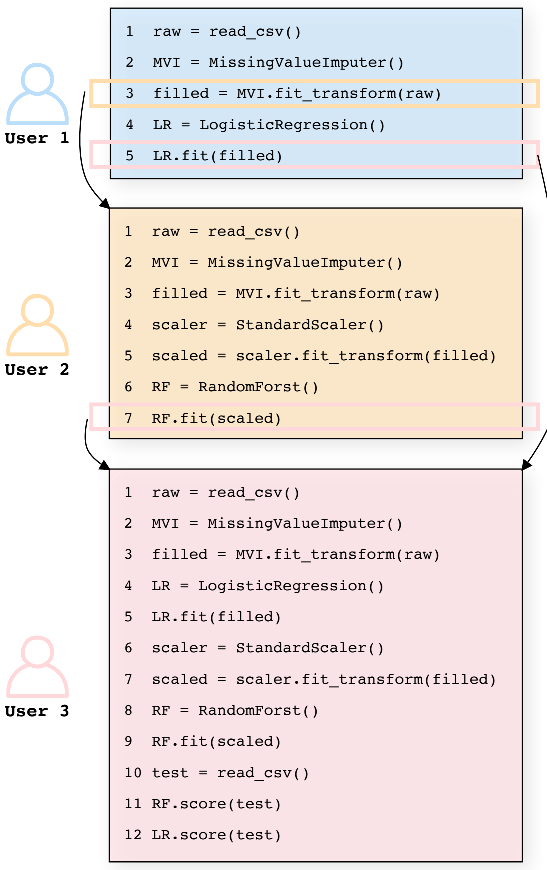
The life cycle of ML applications typically starts with an exploratory analysis over one or a set of raw datasets. The goal of this exploratory analysis is to discover patterns in the dataset and design candidate pipelines for solving ML tasks. The exploratory phase is rarely an individual effort as it requires knowledge in varying fields. Thus, it is a common practice that a group of scientists and practitioners work collaboratively to design ML pipelines. We use the term *Collaborative Setting* to describe this phase of the ML application life cycle where several users work together. We refer to the workloads in the collaborative setting as *Collaborative Workloads*.

The result of the collaborative workloads is typically a candidate pipeline. Data scientists deploy this candidate pipeline into an environment for answering prediction queries. Such a pipeline, however, must be retrained as new data arrives. We refer to this setting, where after deployment, a pipeline is continuously retrained as *Single-pipeline Deployment Setting*. *Single-pipeline Workloads* refer to the workloads in the single-pipeline deployment setting.

In many cases, one pipeline may not suffice. For example, in A/B testing [8, 9, 10] and ensemble learning [11, 12], multiple pipelines are in production at once. In such scenarios, the pipelines differ in some aspects, e.g., ML model type or its hyperparameters, components of the ML pipeline, or schedule of the model training (data interval and training frequency). We name this setting *Multi-pipeline Deployment Setting* and refer to the workloads in this setting as *Multi-pipeline Workloads*.

Figure 1.1 shows the life-cycle of ML applications and the workload settings. The result of the collaborative workloads is a candidate pipeline which is then utilized in a deployment environment. In some use cases, such as ensemble learning or A/B testing, multiple pipelines are in the deployment environment in parallel. In such scenarios, the result of the collaborative workloads is a set of candidate pipelines.

1.1.1 Collaborative Setting



```
1  raw = read_csv()
2  MVI = MissingValueImputer()
3  filled = MVI.fit_transform(raw)
4  LR = LogisticRegression()
5  LR.fit(filled)
```

```
1  raw = read_csv()
2  MVI = MissingValueImputer()
3  filled = MVI.fit_transform(raw)
4  scaler = StandardScaler()
5  scaled = scaler.fit_transform(filled)
6  RF = RandomForst()
7  RF.fit(scaled)
```

```
1  raw = read_csv()
2  MVI = MissingValueImputer()
3  filled = MVI.fit_transform(raw)
4  LR = LogisticRegression()
5  LR.fit(filled)
6  scaler = StandardScaler()
7  scaled = scaler.fit_transform(filled)
8  RF = RandomForst()
9  RF.fit(scaled)
10 test = read_csv()
11 RF.score(test)
12 LR.score(test)
```

Figure 1.2: Example of a collaborative workload. Three users combine their efforts to train and evaluate machine learning pipelines.

Designing effective ML applications requires knowledge in statistics, big data technologies, ML systems, and domain expertise. Therefore, the development of ML applications is rarely an individual effort. To foster collaborative efforts, many platforms utilize containerized environments, such as docker [7], and Jupyter Notebooks [13]. As a result, such platforms can provide quick access to resources for writing and executing data science scripts. Furthermore, they enable the publishing and sharing of the scripts with other data scientists.

Figure 1.2 shows an example of a collaborative workload. User 1 designs a simple pipeline with a logistic regression model. After studying the script of User 1, User 2 performs the same missing value imputation (Line 3) as User 1, scales the data (Lines 4 and 5), and trains a random forest model. User 3 would like to compare the performances of models trained by User 1 and User 2 on a custom test dataset.

Although this approach encourages knowledge sharing, it has one major drawback. To utilize the existing notebooks, one must re-execute them from scratch to recreate the final or partial result in order to continue improving it. This generates large amounts of computational redundancies. Platforms such as Kaggle [14] and Google Colab [15] host many popular notebooks that are executed thousands of times. In some scenarios, e.g., small datasets, the overhead of re-executing the computation is acceptable.

However, the extra overhead hinders productivity as it increases the run time. Furthermore, the increased run time means a higher cost of executing such notebooks.

1.1.2 Single-pipeline Deployment Setting

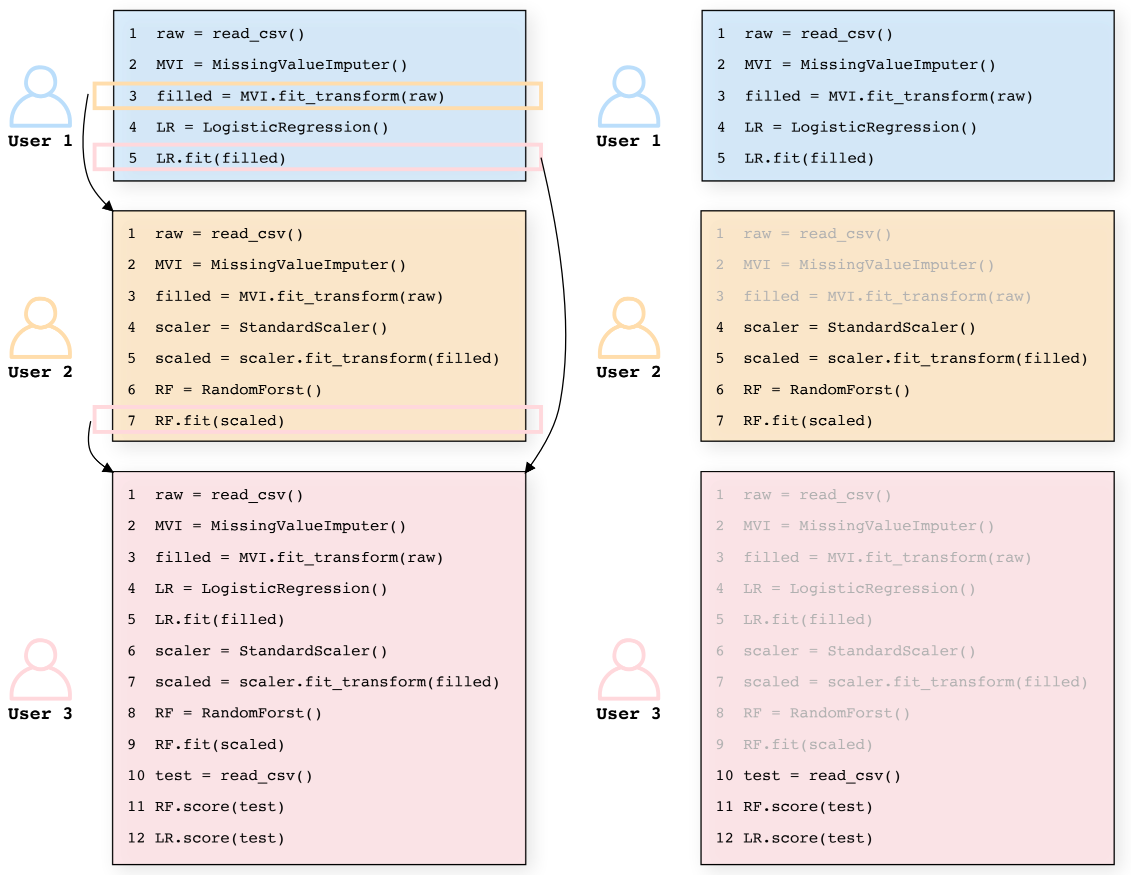
The result of the initial exploratory analysis, model selection, and hyperparameter tuning is an ML pipeline, i.e., the single-pipeline workload. To fully utilize this pipeline, data scientists must deploy the pipeline for answering prediction queries. In many scenarios, new training data continuously arrive at the system. For example, in ad click-through rate prediction [16], after showing an ad to a user, the ad serving system generates new training data based on the click outcome. To maintain the quality of the deployed model, data scientists must update the model by utilizing the new training data. The existing approaches to ensure high-quality models are to utilize online learning [17], periodical retraining [18], or a hybrid of the two [19]. Online learning alone is sensitive to noise and outliers, which may inadvertently negatively impact the model quality. Therefore, to guarantee a high level of quality, existing online learning-only approaches are tailored to specific use cases, which reduces their robustness [20, 16]. As a result, periodical retraining has become the standard approach for single-pipeline workloads. However, periodical retraining generates both data processing redundancy and model training redundancy. Data processing redundancy occurs when the pipeline reprocesses the old data during a retraining process. Model training redundancy is the result of retraining a new model from scratch even though there is an existing model that is trained on older data.

1.1.3 Multi-pipeline Deployment Setting

While deploying one pipeline in a production environment is common in some cases, many use cases require multiple pipelines in parallel, i.e., multi-pipeline workloads. Common use cases of such practice are A/B testing [8, 9, 10], continuous delivery and integration of ML pipelines [21, 22, 23], and ensemble learning [11, 12]. In a multi-pipeline deployment setting, data scientists design several pipelines and deploy all of them to the production environment. In multi-pipeline workloads, the prediction requests are either duplicated to all the pipelines (i.e., ensemble learning) or directed to a specific pipeline based on some criteria (i.e., A/B testing). Furthermore, based on the current performance of the existing pipelines, data scientists may add new pipelines or remove/modify the existing pipelines. When new training data becomes available, the existing models should stay up to date. Therefore, in this setting, it is common to provide a training schedule. The training schedule specifies how often each pipeline should process the data and train the model and what interval to use in each execution. For example, to study the temporal impact of the incoming training data, data scientists design a workload with three pipelines and utilize the following schedules:

1. execute the first pipeline every day on the last 7 days of data,
2. execute the second pipeline every day on the last 14 days of data,
3. and execute the third pipeline every day on the last 30 days of data.

This results in overlapping data processing and model training. Furthermore, deployed pipelines may share similar components. For example, data cleaning [24] and encoding of categorical



```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
LR = LogisticRegression()
LR.fit(filled)
```

```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
scaler = StandardScaler()
scaled = scaler.fit_transform(filled)
RF = RandomForst()
RF.fit(scaled)
```

```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
LR = LogisticRegression()
LR.fit(filled)
scaler = StandardScaler()
scaled = scaler.fit_transform(filled)
RF = RandomForst()
RF.fit(scaled)
test = read_csv()
RF.score(test)
LR.score(test)
```

```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
LR = LogisticRegression()
LR.fit(filled)
```

```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
scaler = StandardScaler()
scaled = scaler.fit_transform(filled)
RF = RandomForst()
RF.fit(scaled)
```

```
raw = read_csv()
MVI = MissingValueImputer()
filled = MVI.fit_transform(raw)
LR = LogisticRegression()
LR.fit(filled)
scaler = StandardScaler()
scaled = scaler.fit_transform(filled)
RF = RandomForst()
RF.fit(scaled)
test = read_csv()
RF.score(test)
LR.score(test)
```

(a) Collaborative Workloads **(b)** Optimized Execution

Figure 1.3: Collaborative workloads and their optimized execution. In an optimized scenario, grey code is skipped and the requested data/model is reused instead.

data are standard techniques in the design of ML pipelines. Therefore, every scheduled pipeline may contain such cleaning or encoding components, which results in further computational redundancy.

1.2 Challenges and Contribution

In this section, for every setting, we first identify the optimization opportunities. Then, we present the challenges and a brief statement of our solution for every challenge.

1.2.1 Collaborative Setting

There are several challenges when optimizing collaborative workloads by materializing and reusing the artifacts, where each artifact is raw or intermediate data and ML models. In an ideal scenario, no repeated executions of operations occur.

Figure 1.3a shows the same collaborative workload as our earlier example. In Figure 1.3b, we show the optimized execution of the scripts for every user. In the optimized execution, the results of the previously executed operations (by any user) are transparently available to new users who plan to execute the same operations. As a result, users do not need to re-execute previously executed operations. However, to achieve such transparent optimizations, we need to address several challenges.

Challenge 1: Management and Storage of Generated Artifacts

In collaborative platforms, where many users write and execute scripts, the number of generated artifacts can grow exponentially. Every script contains tens to hundreds of intermediate data and ML models. The materialization and reuse of such artifacts require proper management and storage of the artifacts as well as their meta-data.

We model an ML workload as a directed acyclic graph (DAG), where vertices represent the artifacts and edges represent the operations in the workload. Each artifact is uniquely identified by its lineage, i.e., the chain of operations that results in the artifact. We store the union of all the workload DAGs in a database, which we refer to as *Experiment Graph*. For every artifact, Experiment Graph also stores a hash of its lineage, which enables a quick lookup. Furthermore, Experiment Graph stores the meta-data of all the artifacts (e.g., column names, size, and model hyperparameters).

Challenge 2: Artifact Materialization Under Limited Storage

The content of the artifacts can be quite large (e.g., raw and transformed data and large ML models). The storage cost of materializing all the artifacts is prohibitive. Selecting the appropriate set of artifacts to materialize, given a budget, is a challenge.

We propose an algorithm for materializing the content of the artifacts given a storage budget. Our materialization algorithm utilizes several metrics, i.e., size, recreation cost, access frequency, and the score of the ML models, to devise a materialization plan. To the best of our knowledge, this is the first work that considers the score of ML models in the materialization plan generation. The intuition behind our strategy is that artifacts leading to high-quality models have a higher probability of future reuse. This intuition stems from our observation of the Kaggle data science platform [14]. In Kaggle, the top publicly available scripts of each competition are accessed and re-executed more frequently than other scripts.

Challenge 3: Overhead of Artifact Reuse

For new workloads, we must devise an execution plan that specifies what artifacts to reuse from Experiment Graph (if they are materialized) and what artifacts to compute. In environments where the rate of incoming workloads is high, the plan generation must incur a negligible overhead.

We propose a linear-time reuse algorithm to find the optimal execution plan for a new ML workload. Our proposed algorithm performs two passes over the workload DAG, i.e., a forward-pass and a backward-pass. In the forward-pass, starting from the source vertices of the DAG (the raw datasets), the algorithm compares the load cost of a materialized artifact with its computation cost and selects the less costly option. In the backward-pass, starting from the terminal vertices, the algorithm prunes any unnecessary vertex. Our reuse algorithm incurs a negligible overhead and handles the high number of incoming ML workloads in large collaborative environments.

1.2.2 Single-pipeline Deployment Setting

Continuous training of deployed ML pipelines on incoming data comprises two main steps. First, the data preprocessing part of the pipeline must process the raw data to generate features. Then, the model training component uses the generated features to train an updated ML model. In single-pipeline deployment settings, there are two main challenges that this thesis addresses.

Challenge 4: Data Processing and Feature Generation

Components of the ML pipeline need to scan the raw data, update their internal statistics, and generate preprocessed features. Periodical statistics computation and feature generation add extra overhead as the size of the collected data grows. Minimizing the overhead of generating the preprocessed features is a challenge.

This thesis proposes *Online Statistics Computation and Dynamic Materialization*. The solution comprises a hybrid data processing approach, i.e., streaming and batch data processing. When new raw data becomes available, the deployed pipeline processes the data as they arrive. The components of the pipeline update their internal statistics in real time and transform the data into preprocessed features. Our solution materializes the preprocessed features for future model training operations. In the presence of a limited storage capacity, we devise a mechanism called dynamic materialization. Dynamic materialization evicts older features and only re-materializes them when needed.

Challenge 5: Retraining of SGD-based Models

Retraining SGD-based models from scratch is wasteful as it ignores existing models. Retraining still requires many iterations until convergence even when model warmstarting is utilized. Furthermore, long training time leads to outdated models. Minimizing the training overhead leads to a lower cost and more up-to-date models.

To alleviate this issue, this thesis devises a technique called *Proactive Training*. An instance of proactive training is analogous to an iteration of the mini-batch stochastic gradient descent algorithm [25]. Proactive training utilizes samples of the generated features, computes the partial gradients, and updates the deployed model in-place using the partial gradients. Proactive training works in close association with dynamic materialization. The sample of the generated features may contain both materialized and evicted features. The dynamic materialization process re-materializes evicted features, combines them with the materialized features, and sends the generated features to proactive training.

Figure 1.4 shows our solution to the discussed challenges for the workloads in the single-pipeline deployment setting. Upon the arrival of new data, our solution utilizes the data preprocessing components of the deployed pipeline to compute and update the internal statistics of the pipeline components (*Online Statistics Computation*). Then, our approach stores the preprocessed features in the data and feature storage unit. To train the model, *Proactive Training* samples the features from the data and feature storage, re-materializes any required features (*Dynamic Materialization*), and finally updates the model.

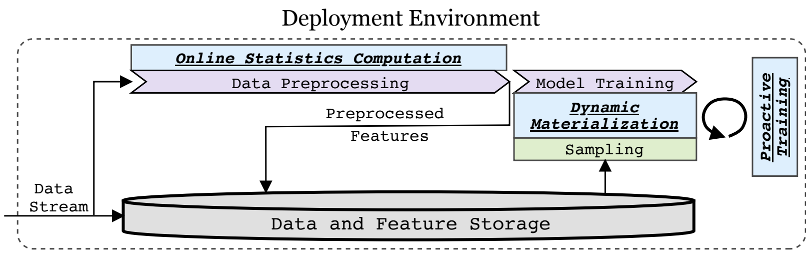


Figure 1.4: Overview of the solution for optimizing single-pipeline workloads.

1.2.3 Multi-pipeline Deployment Setting

To support simultaneous deployment and training of multiple pipelines, we must address several new challenges. For single-pipeline workloads, our solution only materializes the last-level features of the pipeline and uses them to train the model. However, in multi-pipeline workloads, deployed pipelines may share intermediate operators. As a result, there is merit in materializing intermediate data. Furthermore, in multi-pipeline deployment settings, a training schedule accompanies each pipeline. The training schedule specifies the training frequency (e.g., hourly or daily) and the data interval (e.g., last seven days or last month). The introduction of the multiple pipelines and their schedules adds to the complexity of the challenges discussed earlier and introduces new challenges and optimization opportunities.

Challenge 6: Heterogeneity of the Generated Artifacts

Executing ML pipelines generates several types of artifacts, namely, computed statistics of the operators, generated features resulting from an operator, and ML models. Furthermore, due to the dynamic nature of the raw data (streaming data), the generated artifacts are associated with specific time intervals of the data. This heterogeneity of the artifacts renders existing materialization and reuse approaches inapplicable as they do not support complex types.

This thesis devises a unified cost model that applies to every type of generated artifact, i.e., computed statistics, generated features, and ML models. As a result, we can model the computation and loading costs of different artifact types in arbitrary time intervals. The cost model allows us to define materialization and reuse procedures for the artifacts.

Challenge 7: Materialization of the Time-ordered Artifacts

When faced with a limited storage capacity, materializing all the generated artifacts for reuse is prohibitive. Furthermore, in multi-pipeline deployment settings, artifacts are time-ordered and have a corresponding time interval. Given the varying number of the generated artifacts by each pipeline, the number of parallel pipelines, and the different time intervals of the data, finding a subset of the artifacts for materialization is a challenge.

This thesis offers a materialization algorithm that associates the reduction in the cost with the storage overhead. For every generated artifact, the algorithm computes a *benefit score*. The benefit score directly correlates to the reduction in time and inversely correlates to the storage size. Furthermore, the algorithm adapts to the changing workload characteristics, i.e., the pipelines and their schedules (frequency and time interval), and deletion, creation, or modification of the pipelines.

Challenge 8: Variety of Execution Plans

Given the set of pipelines in the workload and the existing materialized artifacts, there are different ways of executing (i.e., execution plans) the pipelines. Each execution plan incurs a different cost. Finding the optimal execution plan—based on the cost model—is challenging as the number of possible plans is large.

Inspired by multi-query optimization techniques, this thesis proposes a reuse and plan generation algorithm that generates the optimal execution plan. The execution plan combines all the individual pipelines into a directed acyclic graph (DAG). The vertices of the DAG—besides the operator—also contain the time interval of the data. Furthermore, the vertices specify if the result is already materialized and can be reused. The complexity of the task arises from the fact that artifacts are defined over intervals. As a result, reusing a materialized artifact in a given interval requires the result to be merged with the portion of the artifact that was computed. Thus, the algorithm also has the task of merging intervals of the artifacts.

Challenge 9: Data Distribution Change

The distribution of the incoming data may change. In such a scenario, reusing artifacts without considering the change in the distribution may generate incorrect results. Identifying what artifacts can be reused in the presence of a distribution change and adapting the execution plan to the distribution change is a challenge.

The reuse and plan generation algorithm reacts to the changes in the distribution. In the presence of a distribution change, the algorithm invalidates artifacts that are affected by the changes in the distribution. The algorithm scans the generated DAG and only allows reuse for vertices that do not have preceding statistics-based operators. Computed statistics may change when the distribution of the data changes. Thus, artifacts resulting from statistics-based operators are outdated, and reusing them may generate incorrect results.

Figure 1.5 shows an overview of our solution for optimizing the workloads in the multi-pipeline deployment setting. The input to the materialization algorithm is the set of deployed pipelines and their training schedules. The output of the materialization algorithm is a materialization plan. The materialization plan contains the intervals of the artifacts that should be materialized. The optimizer receives the materialization plan. Before every execution, the *distribution monitor* checks if there is a distribution change. The *reuse algorithm*—given the available set of materialized artifacts and the presence/absence of a distribution shift—generates the optimal execution plan.

1.3 Impact of Thesis Contributions

The primary contributions of the author and the results of the thesis corresponding to the three studied settings are presented in the following publications.

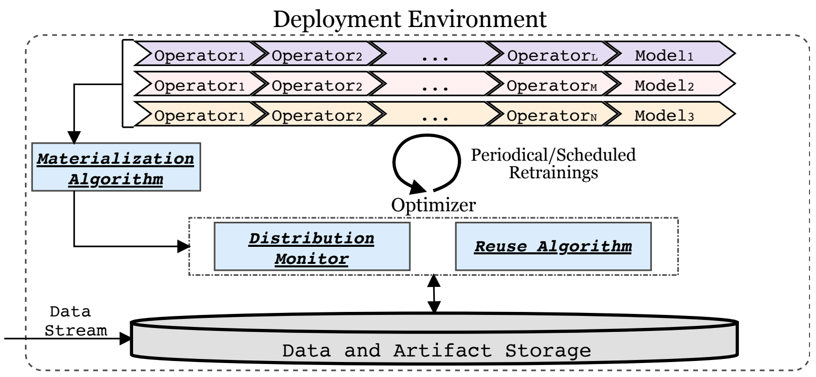

Deployment Environment
Operator1 Operator2 ... OperatorL Model1
Operator1 Operator2 ... OperatorM Model2
Operator1 Operator2 ... OperatorN Model3
Materialization Algorithm
Periodical/Scheduled Retrainings
Optimizer
Distribution Monitor
Reuse Algorithm
Data Stream
Data and Artifact Storage

Figure 1.5: Overview of the solution for optimizing multi-pipeline workloads.

1. **Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl**. "Continuous Deployment of Machine Learning Pipelines." International Conference on Extending Database Technology (EDBT), 2019.
2. **Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl**. "Optimizing Machine Learning Workloads in Collaborative Environments." In Proceedings of ACM International Conference on Management of Data (SIGMOD), 2020
3. **Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl**. "Materialization and Reuse Optimizations for Production Data Science Pipelines." In Proceedings of ACM International Conference on Management of Data (SIGMOD), 2022

1.4 Structure of the Thesis

Chapter 2. Chapter 2 introduces the collaborative setting and highlights its importance in advancing state of the art in machine learning. Chapter 2 covers the publication *"Optimizing Machine Learning Workloads in Collaborative Environments"*. In the chapter, we show the most common approach for enabling collaboration and executing collaborative workloads. Then, we discuss the sources of inefficiency and redundancy in collaborative workloads. Finally, we propose materialization and reuse algorithms to optimize the execution of ML workloads and perform several experiments to show the benefit of our proposed techniques.

Chapter 3. Chapter 3 introduces the single-pipeline deployment setting. Chapter 3 covers the publication *"Continuous Deployment of Machine Learning Pipelines"*. The most common existing approaches, i.e., online learning and periodical retraining, have several disadvantages. In this chapter, we first outline the issues with existing approaches. Then, we introduce *Online Statistics Computation*, *Dynamic Materialization*, and *Proactive Training* to optimize the execution of workloads in the single-pipeline deployment setting.

Chapter 4. In many use cases, the deployment of a single pipeline is not adequate. For example, in A/B testing or ensemble learning, several pipelines are in production at once. In Chapter 4, we introduce the multi-pipeline deployment setting. In the chapter, we first present the existing problems with retraining multi-pipeline workloads. Then, we propose novel materialization and reuse algorithms to take advantage of the pipeline similarities and optimize the execution process. Chapter 4 covers the publication *"Materialization and Reuse Optimizations for Production Data Science Pipelines"*.

Chapter 5. We introduce related work in Chapter 5.

Chapter 6. Chapter 6 lists additional research contributions of the author. These contributions are made while working on the thesis but are not covered in previous chapters.

Chapter 7. Chapter 7 concludes the thesis and outlines future work.

2

Optimizing ML Workloads in Collaborative Settings

In this chapter, we introduce the collaborative setting and the collaborative workloads. Effective collaboration among data scientists results in high-quality and efficient machine learning (ML) applications. In a typical collaborative setting, data scientists publish their ML scripts to make them available for others. In turn, other data scientists utilize the published scripts to improve upon them. This process typically involves re-execution (either fully or partially) of the published scripts to recreate or improve the results. This introduces many redundant data processing and model training operations in collaborative workloads. Reusing the data generated by the redundant operations leads to the more efficient execution of future workloads. However, existing collaborative environments lack a data management component for storing and reusing the result of previously executed operations.

In this chapter, we present a solution to optimize the execution of the collaborative workloads by reusing previously performed operations and their results. We utilize a so-called Experiment Graph (EG) to store and manage the artifacts, i.e., raw and intermediate data or ML models, as vertices and operations of ML workloads as edges. In theory, the size of EG can become unnecessarily large, while the storage budget might be limited. At the same time, for some artifacts, the overall storage and retrieval cost might outweigh the recomputation cost. To address this issue, we propose two algorithms for materializing artifacts. The algorithms estimate the likelihood of future reuse for every artifact and materialize the ones with the highest likelihood. Given the materialized artifacts inside EG, we devise a linear-time reuse algorithm to find the optimal execution plan for incoming ML workloads. Our reuse algorithm only incurs a negligible overhead and can accommodate the high number of incoming ML workloads in collaborative environments. To evaluate our optimization techniques, we implement a system prototype. The prototype includes Experiment Graph, our materialization algorithms, and our reuse algorithm. We utilize real workloads from a Kaggle competition [26] and OpenML [27] to evaluate our solution. Our experiments show that we improve the run time by one order of magnitude for repeated workloads execution and 50% for the execution of the modified workloads in collaborative settings.

2.1 Motivation

Machine learning (ML) plays an essential role in industry and academia. Developing effective ML applications requires knowledge in statistics, big data, and ML systems as well as domain expertise. Therefore, ML application development is not an individual effort and requires collaborations among different users. Recent efforts attempt to enable easy collaboration among users. Platforms such as AzureML [28], Kaggle [14], and Google Colab[15] provide a collaborative environment where users share their scripts and results using Jupyter notebooks [13]. Other platforms such as OpenML [29] and ModelDB [30] enable collaboration by storing ML pipelines, hyperparameters, models, and evaluation results in experiment databases [31].

In such a collaborative setting, the platforms (e.g., Kaggle and Google Colab) typically act as execution engines for ML workloads, i.e., ML scripts. Some platforms also store artifacts. Artifacts refer to raw or intermediate datasets or ML models. By automatically exploiting the stored artifacts, we can improve the execution of the collaborative workloads by skipping redundant operations. However, in existing collaborative settings, there is no automatic management of the stored artifacts. Therefore, users have to manually search through the artifacts and incorporate them into their workloads. We identify two challenges that prohibit the existing platforms from automatically utilizing the existing artifacts. First, the quantity and size of the artifacts are large, which renders their storage unfeasible. For example, we observe that three popular ML scripts in a Kaggle competition [26] generate up to 125 GB of artifacts. Second, ML workloads have a complex structure; thus, automatically finding artifacts for reuse is challenging.

2.2 Research Contributions

We propose a solution for optimizing the execution of collaborative workloads, which addresses the two specified challenges. Our solution stores the artifacts with a high likelihood of reappearing in future workloads. Furthermore, our solution organizes the ML artifacts and offers a linear-time reuse algorithm.

We model an ML workload as a directed acyclic graph (DAG), where vertices represent the artifacts and edges represent the operations in the workload. An artifact comprises two components: meta-data and content. Meta-data refers to the column names of a dataframe, hyperparameters of a model, and evaluation score of a model on a testing dataset. Content refers to the actual data inside a dataframe or the weight vector of an ML model. We refer to the union of all the workload DAGs as *Experiment Graph* (EG), which is available to all the users in the collaborative environment. The size of the artifact meta-data is small. Thus, EG stores the meta-data of all the artifacts. The content of the artifacts is typically large. Therefore, there are two scenarios where storing the content of the artifacts in EG is not suitable, i.e., storage capacity is limited and recomputing an artifact is faster than storing/retrieving the artifact. We propose two novel algorithms for materializing the content of the artifacts given a storage budget. Our materialization algorithms utilize several metrics such as the size, recreation cost, access frequency, operation run time, and the score of the ML models to decide what artifacts to store. To the best of our knowledge, this is the first work that considers the score of ML models in the materialization decision.

To optimize the execution of the incoming ML workloads, we propose a linear-time reuse algorithm that decides whether to retrieve or recompute an artifact. Our reuse algorithm receives a workload DAG and generates an optimal execution plan that minimizes the total execution cost, i.e., the sum of the retrieval and the computation costs. However, for some ML model artifacts, due to the stochasticity of the training operations and differences in hyperparameters, we cannot reuse an existing model. Instead, we warmstart such training operations with a model artifact from EG. Model warmstarting increases the convergence rate resulting in faster execution time of the model training operations.

In summary, we make the following contributions:

- We propose a solution to optimize the execution of ML workloads in collaborative settings. At the center of the proposed solution, is Experiment Graph, a collection of the artifacts and operations of the ML workloads.
- We propose novel algorithms for materializing the artifacts based on their likelihood of future reuse. The algorithms consider run time, size, and the score of ML models.
- We propose a linear-time reuse algorithm for generating optimal execution plans for the ML workloads.

The rest of this chapter is organized as follows. In Section 2.3, we provide some background information. We introduce our collaborative workload optimizer in Section 2.4. In Section 2.5, we discuss our data model and programming API. In Sections 2.6 and 2.7, we introduce the materialization and reuse algorithms. In Section 2.8, we present our evaluations. Finally, we conclude this chapter in Section 2.9.

2.3 Background and Use Case

In this section, we first present a typical collaborative environment. Then, we discuss a motivating example.

Collaborative Environment for Data Science. A typical collaborative environment consists of a client and server. Users write a script to fetch datasets from the server, analyze the data, and train ML models. Then, the client executes the script. Although the client can be a single machine, users typically utilize Jupyter notebooks [13] to write and execute their scripts in isolated containers [7] within the server itself [14, 15, 32]. Users can publish the results and the scripts on the server. Isolated execution environments enable better resource allocation for running scripts.

Motivating Example. Kaggle is a collaborative environment that enables users and organizations to publish datasets and organize ML competitions. In every competition, the organizer defines a task. Users submit their solutions as ML scripts. Kaggle utilizes docker containers, called kernels, to execute user workloads.

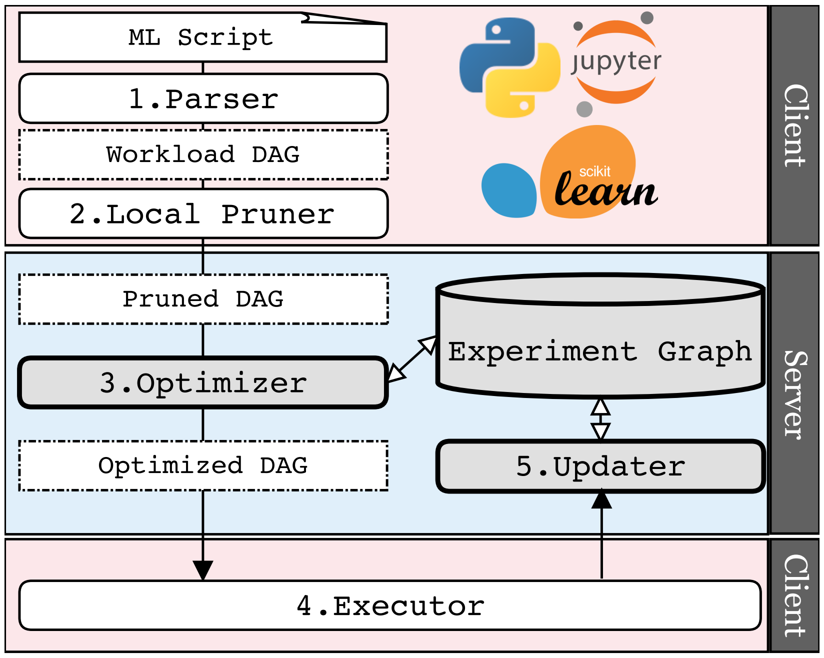


Figure 2.1: Workflow and architecture of the collaborative workload optimizer.

For example, let's consider the competition *Home Credit Default Risk*[1]. The task is to train a classification model to predict whether clients can repay their loans. There are a total of 9 datasets, 8 for training and 1 for evaluation, with a total size of 2.5 GB. The goal of the submitted workloads is to train an ML model that maximizes the area under the ROC curve, which measures how well a classifier works. Three of the most popular submitted workloads are copied and edited by different users more than 7000 times [33, 34, 35]. The three workloads produce 100s of data artifacts and several ML models with a total size of 125 GB. The execution time of each workload is between 200 to 400 seconds.

Kaggle does not store the artifacts, nor does it offer automatic reuse. Therefore, every time a user executes these workloads (or a modified version of them), Kaggle runs them from scratch. Our system, which stores the artifacts and reuses them later, can save hundreds of hours of execution time only for the three workloads in the motivating example. In the next sections, we show how we selectively store artifacts, given a storage budget, and how we quickly find the relevant artifacts for reuse.

2.4 Collaborative ML Workload Optimizations

In this section, we present our system for optimizing collaborative ML workloads. Figure 2.1 shows the architecture of our system, which comprises a client and server component. The client parses the user workload into a DAG (Step 1) and prunes the workload DAG (Step 2). The server receives the workload DAG and utilizes our reuse algorithm to optimize the DAG (Step 3) and returns it to the client. Finally, the client executes the optimized DAG (Step 4) and prompts the server to update Experiment Graph and store the artifacts of the workload DAG (Step 5). This architecture enables us to integrate our system into the existing

[1] `https://www.kaggle.com/c/home-credit-default-risk/`

collaborative environments without requiring any changes to their workflow. The client and server can run within a single cloud environment where each client is an isolated container.

```
 1 import wrapper_pandas as pd
 2 from wrapper_sklearn import svm
 3 from wrapper_sklearn.feature_selection import SelectKBest
 4 from wrapper_sklearn.feature_extraction.text import CountVectorizer
 5
 6 train = pd.read_csv('train.csv') # [ad_desc,ts,u_id,price,y]
 7 ad_desc = train['ad_desc']
 8 vectorizer = CountVectorizer()
 9 count_vectorized = vectorizer.fit_transform(ad_desc)
10 selector = SelectKBest(k=2)
11 t_subset = train[['ts','u_id','price']]
12 y = train['y']
13 top_features = selector.fit_transform(t_subset, y)
14 X = pd.concat([count_vectorized,top_features], axis = 1)
15 model = svm.SVC().fit(X, y)
16 print model # terminal vertex
```

Listing 2.1: Example of a machine learning script.

2.4.1 Client Components

ML Script and Parser. We design an extensible DSL, which enables integration with Python data analysis and ML packages, such as Pandas [36] and scikit-learn [37]. After invoking a script, the parser generates a DAG. Listing 2.1 shows an example of a workload script. The workload processes a dataset of ads description and trains a model to predict if an ad leads to a purchase. Our system supports both long-running Python scripts and interactive Jupyter notebooks.

Workload DAG. In our DAG representation, vertices are the artifacts, i.e., raw or preprocessed data and ML models, and edges are the operations. A workload DAG has one or more source vertices representing the raw datasets. A workload DAG also contains one or more terminal vertices. Terminal vertices are the output of the workload. For example, a terminal vertex is a trained ML model or aggregated data for visualization. Requesting the result of a terminal vertex triggers the optimization and execution of the workload DAG. Figure 2.2 shows the workload DAG constructed of the code in Listing 2.1. In the Figure, the terminal vertex is the result of the print statement on Line 16 in Listing 2.1.

Local Pruner. Once the user requests the result of a terminal vertex, the client prunes the DAG before sending it to the server. The pruner identifies edges that are not in the path from source to terminal and edges with their endpoint vertex already computed. The latter is very common in interactive workloads since every cell invocation in Jupyter notebooks computes some of the vertices. As a result, in future cell invocations, previously executed operations can be skipped. Note that the pruner does not remove the edge from the DAG and only marks them as inactive. For example, in Figure 2.2, if *t_subset* is computed, the local pruner marks

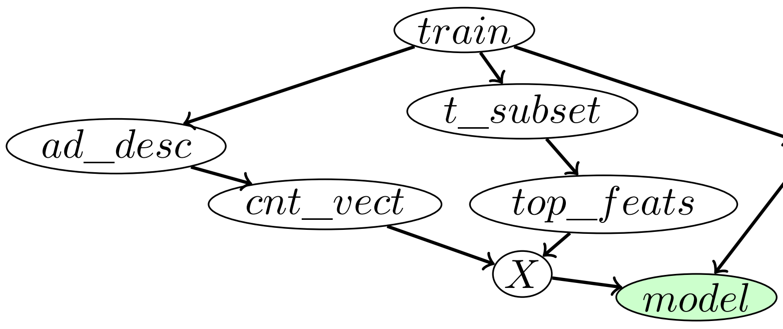


Figure 2.2: Workload DAG constructed from the Listing 2.1. The highlighted node shows a terminal vertex.

the edge between *train* and *t_subset* as inactive. After the pruning, the client sends the DAG to the server.

Executor. After the server optimizes a workload DAG, the executor receives the optimized DAG to execute the operations and returns the result to the user. The executor runs the operations in the optimized DAG in their topological order and returns the result to the user. After the executor completes the execution of a workload DAG, it annotates the DAG vertices with compute-time and sizes before sending it to the updater for storage.

2.4.2 Server Components

Experiment Graph (EG). EG is the union of all the executed workload DAGs, where vertices represent the artifacts and edges represent the operations. Every vertex in EG has the attributes *frequency*, *size*, and *compute_time*, representing the number of workloads an artifact appeared in, the storage size, and the compute-time of the artifact, respectively. Every vertex in EG carries the meta-data of the artifact it represents. For datasets, the meta-data includes the name, type, and size of the columns. For ML models, the meta-data includes the name, type, hyperparameters, and the evaluation score of the model. To save storage space, EG does not contain the content, i.e., underlying data and model weights, of all the artifacts. The updater component decides whether to store the content of an artifact.

EG maintains a list of all the source vertices that it contains. Furthermore, every edge in the graph stores the hash of the operation it represents. Therefore, given a workload DAG, EG quickly detects if it contains the artifacts of the workload DAG by traversing the edges starting from the source.

Optimizer. The optimizer receives the workload DAG from the client and queries EG for materialized artifacts. Then, the optimizer utilizes our reuse algorithm to generate an optimized DAG by retrieving the optimal subset of the materialized vertices from EG. The optimized DAG guarantees to incur the smallest cost, i.e., the transfer cost of the materialized artifacts plus the execution cost of the operations.

Updater. The updater receives the executed DAG from the client. The vertices in the executed DAG contain the size and compute-time of the artifacts they represent. The updater performs the three following tasks. First, it stores any source artifact, both the meta-data and the content, that is not in EG. This is to ensure that EG contains every raw dataset. Second, it updates EG to include all the vertices and edges of the executed DAG. If EG already contains a vertex, the updater increases its frequency. Lastly, by utilizing our novel materialization algorithms, the updater stores the content of a selected set of artifacts, i.e., the output of the materialization algorithms. Note that EG contains the meta-data of all the artifacts, including the unmaterialized artifacts.

2.4.3 Improved Motivating Example

By utilizing our collaborative workload optimizer, we can improve the execution of the workloads in our motivating example. We maintain an EG for the Home Credit Default Risk competition. After users publish their workload scripts on Kaggle, other users will read, re-run, or modify the scripts. The updater component of our system stores the artifacts with a high likelihood of reuse into EG. Our optimizer generates efficient workloads by querying EG for materialized artifacts and transforming the workload DAG into a more optimized DAG. We highlighted three workloads that were copied and modified 7000 times. Optimizing these workloads saves hundreds of hours of execution time, which reduces the required resources and operation cost of Kaggle.

2.5 Representation and Programming Interface

In this section, we first introduce our graph data model and then present the APIs of our system.

2.5.1 Graph Data Model

We represent an ML workload as a directed acyclic graph (DAG). Here, we describe the details of the DAG components (nodes and edges), the construction process, and our approach for representing conditional and iterative programs.

Nodes. Nodes in the graph represent data. We support three types of data: (1) *Dataset*, which has one or more columns of data, analogous to dataframe objects [36], (2) *Aggregate*, which contains a scalar or a collection, and (3) *Model*, which represents a machine learning model.

Edges. An edge (v_1, v_2) represents the operation that generates node v_2 using node v_1 as input. There are two types of operations. (1) *Data preprocessing operations*, which include data transformation and feature engineering operations that generate either a Dataset (e.g., map, filter, or one-hot encoding) or an Aggregate (e.g., reduce). (2) *Model training operations*, which generate a Model. A Model is used either in other feature engineering operations, e.g., PCA model, or to perform predictions on a test dataset.

Multi-input Operations. To represent operations that have multiple inputs (e.g., join), we use a special node type, which we refer to as a *Supernode*. Supernodes do not contain underlying data and only have incoming edges from the input nodes. The outgoing edge from a supernode represents the multi-input operations.

DAG Construction. The DAG construction starts with a source vertex (or multiple source vertices) representing the raw data. For every operation, the system computes a hash based on the operation name and its parameters. In interactive workloads (i.e., Jupyter Notebooks), the DAG can continue to grow after an execution.

Conditional Control Flows. To enable support for conditional control flows, we require the condition statement of the iteration or if-statement to be computed before the control flow begins. This is similar to how Spark RDDs [38] handle conditional control flows. Furthermore, we represent an iterative ML training operation as one operation (edge) inside the DAG and do not materialize fine-grained model iteration data (i.e., the model parameters after every iteration). Such a materialization strategy is useful for model diagnosis queries [39], where users need to analyze the parameters of a model after each iteration. However, such scenarios are not the focus of our work.

2.5.2 Parser and API

We use Python as the language of the platform. This allows seamless integration to third-party Python libraries.

Parser and Extensibility. Our platform provides two levels of abstraction for writing ML workloads. The code in Listing 2.1 (Section 2.4) shows the high-level abstraction, which exposes an identical API to the Pandas and scikit-learn models. The parser translates the code to the lower level abstraction, which directly operates on the nodes of the graph and creates the DAG components.

```python
class Operation(object):
    def __init__(self, name, return_type, params):
      self.name = name
      self.return_type = return_type
      self.params = params
    ...

class DataOperation(Operation):
    def __init__(self, name, return_type, params):
        Operation.__init__(name, return_type, params)

    @abstractmethod
    def run(self, underlying_data):
        pass

class TrainOperation(Operation):
    def __init__(self, name, warmstarting, params):
        Operation.__init__(name, Types.Model, params)
        self.warmstarting = warmstarting

    @abstractmethod
    def run(self, underlying_data, warmstarting_candidate=None):
        pass
```

Listing 2.2: Definition of operation classes.

In the lower level abstraction, every node has an `add` method, which receives an operation. There are two types of operations, i.e., `DataOperation` and `TrainOperation`. Listing 2.2 shows the abstraction of base `Operation` as well as the `DataOperation` and `TrainOperation` classes. To define new data preprocessing or model training operations, users must extend the `DataOperation` or `TrainOperation` classes. When defining new operations, users must indicate the name, return type, and the parameters of the operation. Users must also implement a `run` method, which contains the main data processing or model training code. For model training operations, the users must also provide a boolean parameter (`warmstarting`) indicating if the training operation can be warmstarted or not. The return type of a `TrainOperation` is always a Model; thus, users do not need to specify it for the training operations.

Listing 2.3 shows an example of implementing a sampling operation. Users extend the `DataOperation` class (Line 1) and specify the name and return type (Line 3). An instance of the operation with different parameters can then be created (Line 9). Inside the `run` method, users have access to the underlying data and can perform the actual data processing. The parser generates a DAG with the following components: (1) a node, which represents `data_node` on Line 10, (2) an outgoing edge from `data_node` representing the `sample_op` on Line 11, and (3) another node representing `sampled_data_node`, the result of the sampling operation. Once the optimizer returns the optimized DAG, the code inside the `run` method of the `Sample` class is executed. The type of the `underlying_data` argument in the `run` method (Line 5) depends on the type of the input node of the operation. For example, in Listing 2.3, the user is applying the sampling operation to the Dataset node loaded from disk (Line 10); thus, the type of the `underlying_data` is dataframe. For *multi-input operations*, the `underlying_data` argument is an array of data objects, where each item represents one of the input nodes to the multi-input operation. Lastly, since the sample operation must return a Dataset, the parser encapsulates the result of the `run` method inside a Dataset node. The process of extending a model training operation is similar. If a model training operation can be warmstarted, the system will provide a warmstarting candidate model if one is available.

```
 1 class Sample(DataOperation):
 2     def __init__(self, params):
 3         Operation.__init__('sample', Types.Dataset, params)
 4
 5     def run(self, underlying_data):
 6         return underlying_data.sample(n=self.params['n'],
 7                             random_state=self.params['r_state'])
 8
 9 sample_op = Sample(params={'n':1000, 'r_state':42})
10 data_node = Dataset.load('path')
11 sampled_data_node = data_node.add(sample_op)
```

Listing 2.3: Defining and using a new operation.

Program Optimization. To find the optimal reuse plan, our optimizer only requires information about the size of the nodes and the execution cost of the operations. The system captures the execution costs and size of the nodes after executing a workload. As a result, when implementing new operations, users do not need to concern themselves with providing extra information for the optimizer.

Integration Limitations. Our APIs allow integration with other feature engineering packages, such as FeatureTools [40], and ML frameworks, such as TensorFlow [41]. However, our optimizer is oblivious to the intermediate data that are generated inside the third-party system. As a result, our optimizer only offers materialization and reuse of the final output of the integrated system.

2.6 Artifact Materialization

Depending on the number of executed workloads, the generated artifacts may require a large amount of storage space. For example, the three workloads in our motivating example generate up to 125 GB of artifacts. Moreover, depending on the storage and retrieval costs of the artifacts from EG, sometimes it may be less costly to recompute an artifact from scratch. In this section, we introduce two algorithms for materializing the artifacts. The goal of the algorithms is to materialize artifacts with a high likelihood of future reuse while ensuring the storage does not surpass the recomputation cost. The first algorithm (Section 2.6.2) utilizes general metrics, i.e., size, access frequency, compute times, and storage cost of the vertices, and an ML-specific metric, i.e., the quality of the ML models, to decide what artifacts to materialize. The second algorithm (Section 2.6.3) extends the first algorithm and considers any overlap between the artifacts, i.e., a data column appearing in multiple artifacts.

Notations. We use the following notations in this section. Graph $G_E = (V, E)$ is Experiment Graph, where V represents the set of artifacts and E represents the set of operations. We use the terms artifact and vertex interchangeably. Each vertex $v \in V$ has the attributes $\langle f, t, s, mat \rangle$. f, t, and s refer to the frequency, computation time, and size while $mat = 1$ indicates v is materialized and 0 otherwise. We also define the set of all ML models in G_E as:

$$M(G_E) = \{v \in V \mid v \text{ is an ML model}\}$$

and the set of all reachable ML models from vertex v as:

$$M(v) = \{m \in M(G_E) \mid \text{there is path from } v \text{ to } m\}$$

Assumptions. We assume there exists an evaluation function that assigns a score to ML models. This is a reasonable assumption as the success of any ML application is measured through an evaluation function. For instance, our motivating example uses the area under the ROC curve for scoring the submitted workloads. In EG, any vertex that represents an ML model artifact contains an extra attribute, q $(0 \leq q \leq 1)$, representing the quality of the model.

2.6.1 Materialization Problem Formulation

Existing work proposes algorithms for the efficient storage of dataset versions and their storage and recomputation trade-off [42]. The goal of the existing algorithms is to materialize the artifacts that result in a small recomputation cost while ensuring the total size of the materialized artifacts does not exceed the storage capacity. However, two reasons render the existing algorithms inapplicable to our artifact materialization problem. First, existing approaches do not consider the performance of ML workloads, i.e., the quality of ML models when materializing artifacts. Second, existing solutions do not apply to collaborative environments, where the rate of incoming workloads is high. Here, we formulate the problem of artifact materialization as a multi-objective optimization problem. The goal of artifact materialization is to materialize a subset of the artifacts that minimizes the weighted recomputation cost while maximizing the estimated quality.

Weighted Recreation Cost Function (WC). The first function computes the weighted recreation cost of all the vertices in the graph:

$$WC(G_E) = \sum_{v \in V} (1 - v.mat) \times v.f \times v.t$$

Intuitively, the weighted recreation cost computes the total execution time required to recompute the vertices while considering their frequencies. Materialized artifacts incur a cost of zero. Unmaterialized artifacts incur a cost equal to their computation time multiplied by their frequencies.

Estimated Quality Function (EQ). EQ computes the estimated quality of all the materialized vertices in the graph. To compute EQ, we first define the potential of a vertex:

$$p(v) = \begin{cases} 0, & \text{if } M(v) = \emptyset \\ \max\limits_{m \in M(v)} m.q, & \text{otherwise} \end{cases}$$

Intuitively, the potential of a vertex is equal to the quality of the best reachable model from the vertex. Note that vertices that are not connected to any model have a potential of 0. Now, we define the estimated quality function as:

$$EQ(G_E) = \sum_{v \in V} v.mat \times p(v)$$

Multi-Objective Optimization. Given the two functions, we would like to find the set of vertices to materialize, which minimizes the weighted recreation cost function and maximizes the estimated quality function under limited storage size, $\mathcal{B}$. For ease of representation, we instead try to minimize the inverse of the estimated quality function. We formulate the optimization problem as follows:

$$\begin{aligned} & minimize(WC(G_E), \frac{1}{EQ(G_E)}), \\ & \text{subject to: } \sum_{v \in V} v.mat \times v.s \leq \mathcal{B} \end{aligned} \tag{2.1}$$

Existing work proves that minimizing the recreation cost alone is an NP-Hard problem [42]. While there are different approximate strategies for solving multi-objective optimization problems [43], they are time-consuming, which renders them inappropriate to our setting, where new workloads are constantly executed. Execution of every workload results in an update to EG, which in turn requires a recomputation of the materialized set. As a result, existing solutions to multi-objective optimization problems are not suitable for artifact materializations of EG.

2.6.2 ML-Based Greedy Algorithm

We propose a greedy heuristic-based algorithm to solve the optimization problem. Our approach is based on the utility function method for solving multi-objective optimizations [44], where we combine the weighted recreation cost and the estimated quality. Our algorithm selects vertices with the largest utility in a greedy fashion.

Algorithm 1: Artifacts materialization algorithm.

```
   Input: $G_E(V, E)$ experiment graph, $\mathcal{B}$ storage budget
   Output: $\mathcal{M}$ set of vertices to materialize
1  $S := 0$;                              // size of the materialized artifacts
2  $\mathcal{M} := \emptyset$;                                  // materialized set
3  $PQ :=$ empty priority queue;
4  for $v \leftarrow V$ do
5      if $v.mat = 0$ then
6          $v.utility := \mathcal{U}(v)$;
7          $PQ.insert(v)$;                               // sorted by utility
8  while $PQ.not\_empty()$ do
9      $v := PQ.pop()$;                           // vertex with max utility
10     if $S + v.s \leq \mathcal{B}$ then
11         $\mathcal{M} := \mathcal{M} \cup v$;
12         $S := S + v.s$;
13 return $\mathcal{M}$;
```

Algorithm 1 shows the details of our method for selecting the vertices to materialize. For every non-materialized vertex, we compute the utility value of the vertex (Lines 4-7). Then, we start materializing the vertices, sorted by their utilities, until the storage budget is exhausted (Lines 8-12). The utility function $\mathcal{U}(v)$ combines the potential, recreation cost, and size of a vertex. We design the utility function in such a way that materializing vertices with larger utility values contributes more to minimizing the multi-objective optimization function (Equation 2.1). Before we define $\mathcal{U}(v)$, we need to define 3 functions: the recreation cost of a vertex $C_r(v)$, the cost-size ratio $r_{cs}(v)$, and the load cost of a vertex $C_l(v)$. The recreation cost of a vertex is:

$$C_r(v) = \sum_{v' \in G_v} v'.t$$

where $G_v \subseteq G_E$ is the compute graph of v, i.e., the set of all vertices and edges which one must execute to recreate the vertex v. The compute graph of a vertex always starts at one or more source vertices of EG and ends at the vertex itself. The weighted cost-size ratio is:

$$r_{cs}(v) = \frac{v.f \times C_r(v)}{v.s}$$

which has the unit $\frac{s}{MB}$ and indicates how much time do we spend on computing 1 MB of an artifact. Lastly, $C_l(v)$ is the cost (in seconds) of loading the vertex v from EG. The $C_l(v)$ function depends on the size of the vertex and where EG resides (i.e., in memory, on disk, or in a remote location). We now define the utility function as the linear combination:

$$\mathcal{U}(v) := \begin{cases} 0, & \text{if } C_l(v) \geq C_r(v) \\ \alpha p'(v) + (1-\alpha) r'_{cs}(v), & \text{otherwise} \end{cases} \tag{2.2}$$

, where $p'(v)$ and $r'_{cs}(v)$ are normalized values of $p(v)$ and $r_{cs}(v)$ (i.e., for every vertex divide the value by the total sum). We never materialize a vertex when $C_l(v) \geq C_r(v)$ since recomputing such a vertex is more efficient. Taking the load cost into account enables us to adapt the materialization algorithm to different system architecture types (i.e., single node vs distributed) and storage unit types (i.e., memory or disk). α ($0 \leq \alpha \leq 1$) indicates the importance of potential. For example, when $\alpha > 0.5$, we assign more importance to model quality than weighted cost-size. In collaborative environments, where the goal is to build high-quality models and data exploration is not the main objective, a larger α encourages faster materialization of high-quality models.

Run time and Complexity. We compute the recreation cost and potential of the nodes incrementally using one pass over EG. Thus, the complexity of the materialization algorithm is $\mathcal{O}(|V|)$ where $|V|$ is the number of vertices in EG. The size of EG increases as users execute more workloads. This increases the execution cost of the materialization algorithm. However, we only need to compute the utility for a subset of the vertices. First, we must compute the utility of the vertices belonging to the new workload. The addition of the new vertices affects the normalized cost and potential of other vertices, thus requiring a recomputation. However, we only need to recompute the utility of the materialized vertices and compare them with the utility of the workload vertices. As a result, the complexity of each run of the materialization algorithm is $\mathcal{O}(|W| + |M|)$, where $|W|$ is the number of vertices in the new workload DAG and $|M|$ is the number of the materialized vertices.

2.6.3 Storage-Aware Materialization

Many feature engineering operations operate only on one or a few columns of a dataset artifact; thus, the output artifact may contain some of the columns of the input artifact. Therefore, materializing both the input and output artifacts may lead to many duplicated columns. To reduce the storage cost, we implement a deduplication mechanism. We assign a unique id to every column of the dataset artifacts. To compute the unique id after the execution of an operation, we first determine the columns which are affected by the operation. Then, we use a hash function that receives the operation hash and id of the input column and outputs a new id. Our approach for computing the unique id ensures the following. First, after the execution of an operation, all the columns which are not affected by the operation will carry the same id. Second, two columns belonging to two different dataset artifacts have the same unique id, if and only if, the same operations have been applied to both columns.

We implement a storage manager that takes the deduplication information into account. The storage manager stores the column data using the column id as the key. Thus, ensuring duplicated columns are not stored multiple times.

Greedy Meta-Algorithm. We propose a storage-aware materialization meta-algorithm that iteratively invokes Algorithm 1 (the artifact materialization algorithm). While the budget is not exhausted, we proceed as follows. First, we apply Algorithm 1 to find the set of vertices to materialize. Then, using the deduplication strategy, we compress the materialized artifacts. We then compute the size of the compressed artifacts and update the remaining budget. Using the updated budget, we repeatedly invoke Algorithm 1, until no new vertices are materialized or the updated budget is zero.

2.7 Reuse and Warmstarting

Our collaborative workload optimizer looks for opportunities to reuse existing materialized artifacts of EG and warmstart model training operations. Every artifact of the incoming workload DAG either does not exist in EG, exists in EG but is unmaterialized, or is materialized. For the first two cases, the client must execute the operations of the workload DAG to compute the artifact. However, when the artifact is materialized, we can choose to load or compute the artifact. Both loading and computing an artifact incur costs. In this section, we describe our linear-time reuse algorithm, which selects the optimal subset of the materialized artifacts to reuse.

2.7.1 Reuse Algorithm

Preliminaries and Notations. We refer to the workload DAG as G_W. Every vertex, $v \in G_W$, has a load cost (i.e., $C_l(v)$ defined in Section 2.6). We also define $C_i(v)$ as the computation cost (in seconds) of v given its input vertices (i.e., the parents of the vertex v) in G_W. If an artifact exist in G_E but is not materialized, then we set $C_l(v) = \infty$. For artifacts that do not exist in G_E, $C_l(v)$ and $C_i(v)$ are also set to ∞. Such artifacts have never appeared in any previous workloads; thus, Experiment Graph has no prior information about them. Lastly, if an artifact is already computed inside G_W, such as the source artifacts or pre-computed artifacts in interactive workloads, we set $C_i(v) = 0$, this is because the artifact is already available in the client's memory.

Linear-time Algorithm. Our reuse algorithm comprises two parts: forward-pass and backward-pass. In forward-pass, the algorithm selects the set of materialized vertices to load from Experiment Graph into the workload DAG. The backward-pass prunes any unnecessary materialized vertices before transferring the optimized DAG to the client.

Algorithm 2 shows the details of forward-pass. For every vertex of G_W, we define the recreation cost as the total cost of computing the vertex from the sources. We store the recreation costs of the vertices in the *recreation_cost* map data structure. Before the execution begins, the client always loads the source artifacts from EG completely. Therefore, we set their recreation cost to 0 (Lines 1 and 2). Then, we visit every vertex in its topological order. If the client has already computed a vertex inside G_W, then we set its recreation cost to 0 (Lines 5 and 6). Otherwise, we compute the execution cost of a vertex as the sum of the compute

Algorithm 2: Forward-pass algorithm.

```
Input: $G_W$ workload DAG
Output: $\mathcal{R}$ set of vertices for reuse
for $s \in sources(G_W)$ do
    $recreation\_cost[s] := 0$;
$\mathcal{R} := \emptyset$;
for $v \leftarrow topological\_order(G_W)$ do
    if $v$ computed in $G_W$ then
        $recreation\_cost[v] := 0$;
    else
        $p\_costs := \sum_{p \in parents(v)} recreation\_cost[p]$;
        $execution\_cost := C_i(v) + p\_costs$;
        if $C_l(v) < execution\_cost$ then
            $recreation\_cost[v] := C_l(v)$;
            $\mathcal{R} := \mathcal{R} \cup v$;
        else
            $recreation\_cost[v] := execution\_cost$;
return $\mathcal{R}$;
```

cost of the vertex and the recreation cost of its parents (Lines 8 and 9). We then compare the load cost of the vertex with the execution cost and set its recreation cost to the smaller of the two (Lines 10-14). When the load cost is smaller, the algorithm adds the vertex to the set of reuse vertices. Note that unmaterialized vertices have a load cost of ∞; therefore, the algorithm never loads an unmaterialized vertex. After forward-pass, we must prune the set of reuse vertices to remove any artifact that is not part of the execution path. A vertex $v \in \mathcal{R}$ is not part of the execution path if there exists at least another vertex $v' \in \mathcal{R}$ in every outgoing path starting at v. In backward-pass, we visit every vertex of G_W starting from the terminal vertices. For every vertex, if it belongs to the reuse set ($\mathcal{R}$), we add it to the final solution set ($\mathcal{R}_p$) and stop traversing its parents. Otherwise, we continue traversing the parents of the vertex.

Figure 2.3 shows how our reuse algorithm operates on an example workload DAG. There are 3 source vertices in the workload DAG. The algorithm starts with forward-pass, traversing the graph from the sources (Scenario ①). For materialized vertices with a smaller load cost than the execution cost (i.e., the sum of the compute cost and parent's recreation costs), the algorithm sets the recreation cost to the load cost of the vertex (Scenario ②). For example, for the materialized vertex v_3, the execution cost is $16 + 5 = 21$, which is larger than its load cost of 20 (same scenario applies to vertex v_1). For vertices that exist in EG but are unmaterialized, the algorithm chooses to compute them (Scenario ③). If a vertex is already computed inside the workload DAG, then the algorithm sets its recreation cost to zero (Scenario ④). For materialized vertices with a larger load cost than the execution cost, the algorithm sets the recreation cost to the execution cost (Scenario ⑤). For example, for the materialized vertex v_2, the execution cost is $10 + 5 + 1 = 16$, which is smaller than its load cost of 17. Once the traversal reaches a node that does not exist in EG, the forward-pass stops (Scenario ⑥). At this stage, forward-pass has selected the materialized vertices v_1 and v_3 for reuse. Then, the algorithm starts backward-pass from the terminal vertex (Scenario ⑦). Once the

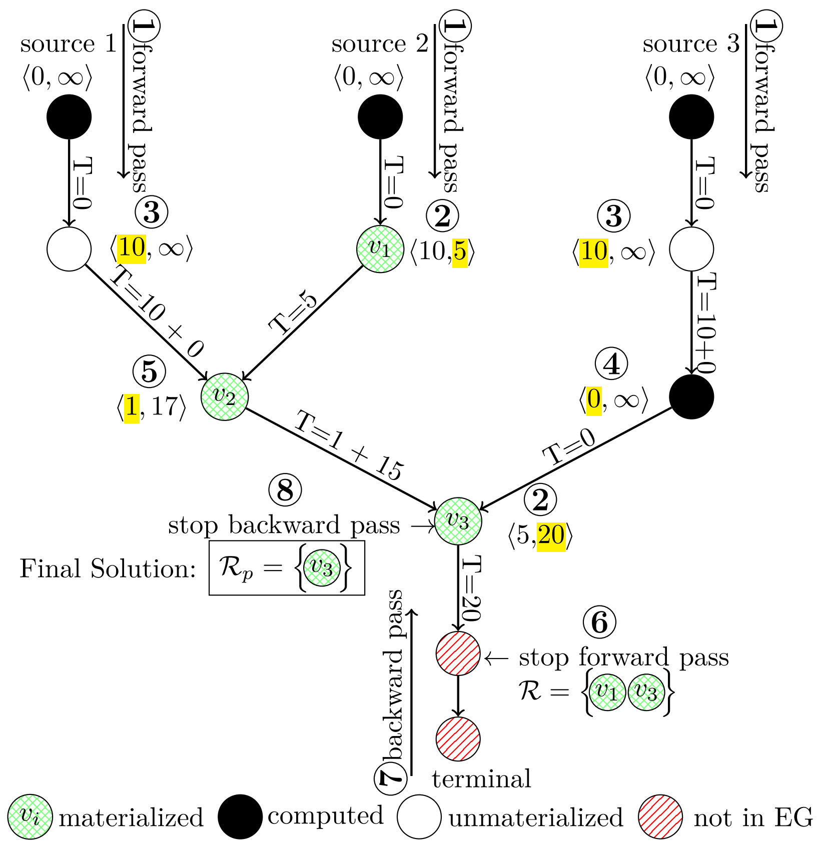


Figure 2.3: An example of the reuse algorithm. Each vertex has the label $\langle C_i(v), C_l(v)\rangle$. T is the recreation cost.

backward-pass visits a materialized vertex, it stops traversing its parents. The backward-pass removes any materialized vertices that it did not visit (Scenario ⑧). For example, since v_3 is materialized, backward-pass stops visiting its parents; thus, it removes v_1 from the final solution.

Complexity. Both forward-pass and backward-pass traverse the workload DAG once, resulting in a maximum of $2 * |V|$ visits. Therefore, our reuse algorithm has a worst-case complexity of $\mathcal{O}(|V|)$. A linear-time algorithm can scale to a large number of workloads, which is typical in collaborative settings such as Kaggle. Furthermore, in our implementation of both forward-pass and backward-pass, we include early-stopping conditions. During forward-pass, if we reach a vertex that does not exist in EG (Scenario ⑥ in Figure 2.3), we stop visiting the successors of the vertex. This is because if a vertex of the workload DAG does not exist in EG, then, none of its successors are in EG. Therefore, there are no reuse opportunities among the successors of the vertex. During backward-pass, if a vertex belongs to the current reuse set ($\mathcal{R}$), we add it to the final solution set and stop traversing its parents (Scenario ⑧ in Figure 2.3). The two early-stopping conditions further reduce the number of vertex visits, which can increase the scalability of our reuse algorithm.

2.7.2 Warmstarting

Many model training operations include different hyperparameters, which impact the training process. Two training operations on the same dataset with different hyperparameters may

Table 2.1: Description of the Kaggle workloads. N is number of the artifacts and S is the total size of the artifacts in GB.

ID	*Description*	N	S
1	A real Kaggle script. It includes several feature engineering operations before training logistic regression, random forest, and gradient boosted tree models [33].	397	14.5
2	A real Kaggle script. It joins multiple datasets, preprocesses the datasets to generate features, and trains gradient boosted tree models on the generated features [34].	406	25
3	A real Kaggle script. It is similar to Workload 2, with the resulting preprocessed datasets having more features [35].	146	83.5
4	A real Kaggle script that modifies Workload 1 and trains a gradient boosted tree with a different set of hyperparameters [45].	280	10
5	A real Kaggle script that modifies Workload 1 and performs random and grid search for gradient boosted tree model using generated features of Workload 1 [46].	402	13.8
6	A custom script based on Workloads 2 and 4. It trains a gradient boosted tree on the generated features of Workload 2.	121	21
7	A custom script based on Workload 3 and 4. It trains a gradient boosted tree on the generated features of Workload 3.	145	83
8	A custom script that joins the features of Workloads 1 and 2. Then, similar to Workload 4, it trains a gradient boosted tree on the joined dataset.	341	21.1

result in completely different models. In such scenarios, we cannot reuse a materialized model in EG instead of a model artifact in the workload DAG. However, we try to warmstart the model training operations using the models in EG to reduce the training time. In warmstarting, instead of randomly initializing the parameters of a model before training, we initialize the model parameters to a previously trained model. Warmstarting has been shown to decrease the total training time in some scenarios [18]. Note that in some scenarios, warmstarting may result in a different trained model. Therefore, we only warmstart a model training operation, when users explicitly request it.

The process of warmstarting is as follows. Once we encounter a model in the workload DAG in forward-pass, we look for warmstarting candidates in EG. A warmstarting candidate is a model that is trained on the same artifact and is of the same type as the model in the workload DAG. When there are multiple candidates for warmstarting, we select the model with the highest quality. Finally, we initialize the model training operation with the selected model.

2.8 Evaluation

In this section, we evaluate the performance of our solution. We first describe the setup of the experiment. Then, we show the end-to-end run time improvement of our solution. Finally, we investigate the effect of the individual contributions, i.e., materialization and reuse algorithms, on the run time and storage cost.

2.8.1 Setup

We execute the experiments on a Linux Ubuntu machine with 128 GB of RAM. We implement a prototype of our system in Python 2.7.12. We implement EG using NetworkX 2.2 [47]. We run every experiment 3 times and report the average.

Baseline and Other Systems. We compare our system with a naive baseline, i.e., executing all the workloads without any optimization, and Helix [6]. Helix is a system for optimizing ML workloads, where users *iterate* on workloads by testing out small modifications until achieving the desired solution. Helix utilizes materialization and reuse of the intermediate artifacts to speed up the execution of ML workloads within a single session. Helix materializes an artifact when its recreation cost is greater than twice its load cost (Algorithm 2 of the Helix paper [6]). To find the optimal reuse plan, Helix reduces the workload DAG into an instance of the project selection problem (PSP) and solves it via the Max-Flow algorithm [48]. In our implementation of Helix reuse, we consulted the authors and followed Algorithm 1 of the Helix paper to transform the workload DAG into PSP. Similar to Helix, we utilized the Edmonds-Karp Max-Flow algorithm [49], which runs in polynomial time ($\mathcal{O}(|V|.|E|^2)$).

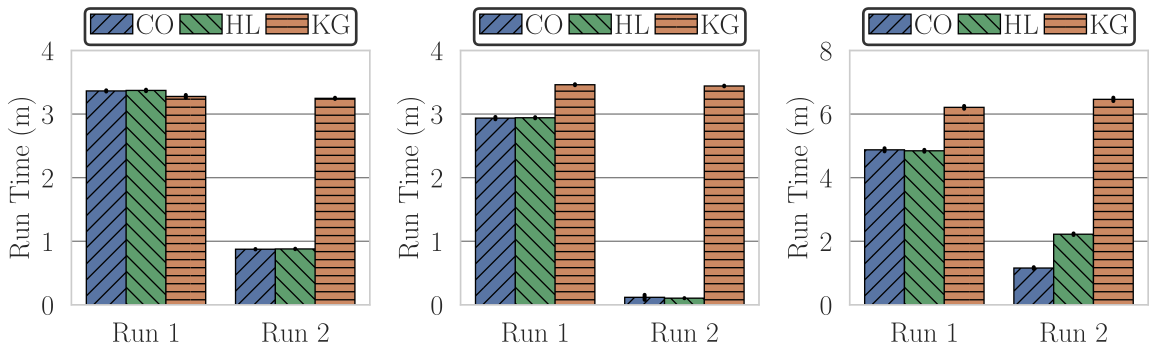
Kaggle Workloads. In the Kaggle workloads, we recreate the use case in Section 2.3. We use eight workloads, which generate 130 GB of artifacts. There are five real and three custom workloads. Table 2.1 shows details of the workloads. There are 9 source datasets with a total size of 2.5 GB. Unless specified otherwise, we use storage-aware materialization with a budget of 16 GB and $\alpha = 0.5$. For Helix, we also set the materialization budget to 16 GB.

OpenML Workloads. In the OpenML workloads, we extracted 2000 runs of scikit-learn pipelines for Task 31 from the OpenML platform [27]. The dataset is small, and the total size of the artifacts after executing the 2000 runs is 1.5 GB. We use the OpenML workloads to show the effects of the model quality on materialization and model warmstarting on run time. Unless specified otherwise, we use storage-aware materialization with a budget of 100 MB and $\alpha = 0.5$.

2.8.2 End-to-end Optimization

In this experiment, we evaluate the impact of our optimizations on the Kaggle workloads. In our motivating example, we describe three workloads (Workloads 1-3 of Table 2.1) that are copied and modified 7,000 times by different users. Therefore, at the very least, users execute these workloads 7000 times.

Figure 2.4 shows the result of repeating the execution of each workload twice. Before the first run, EG is empty. Therefore, the default baseline (KG), Helix (HL), and our collaborative optimizer (CO) must execute all the operations in the workloads. In Workload 1, the run time of CO and HL is slightly larger than KG in the first run. Workload 1 executes two alignment operations. An alignment operation receives two datasets, removes all the columns that do not appear in both datasets, and returns the resulting two datasets. In CO, we need to measure the precise compute cost of every artifact. This is not possible for operations that return multiple artifacts. Thus, we re-implemented the alignment operation, which is less optimized than the baseline implementation. In Workloads 2 and 3, CO and HL outperform KG even in



(a) Workload 1. **(b)** Workload 2. **(c)** Workload 3.

Figure 2.4: Total run time of repeated executions of different Kaggle workloads.

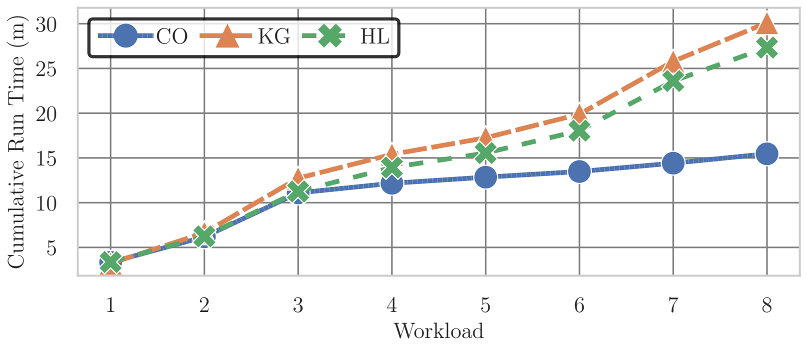


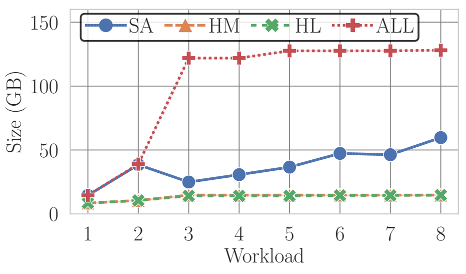
Figure 2.5: Execution of the Kaggle workloads in sequence.

the first run. Both Workloads 2 and 3 contain many redundant operations. The local pruning step of our optimizer identifies the redundancies and only executes such operations once.

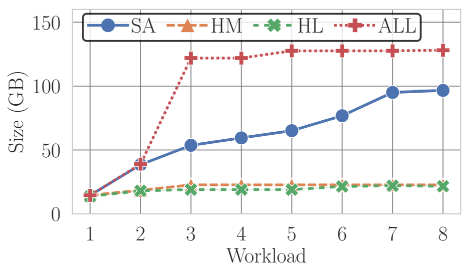
In the second run of the workloads, CO reduces the run time by an order of magnitude for Workloads 2 and 3. Workload 1 executes an external and compute-intensive visualization command that computes a bivariate kernel density estimate. Since our optimizer does not materialize such external information, it must re-execute the operation; thus, resulting in a smaller run time reduction.

HL has a similar performance to CO in Workloads 1 and 2. However, CO outperforms HL in Workload 3. The total size of the artifacts in Workloads 1 and 2 is small. As a result, a large number of artifacts for both HL and CO are materialized. Our reuse algorithm finds the same reuse plan as Helix, therefore, the run times for Workloads 1 and 2 are similar. However, the size of the artifacts in Workload 3 is larger than the budget (i.e., 83.5 GB). The materialization algorithm of HL does not consider the benefit of materializing one artifact over the others and starts materializing the artifacts from the source vertex until the budget is exhausted. As a result, many of the high-utility artifacts that appear towards the end of the workloads are not materialized. The side-effect of the materialization algorithm of HL is visible for Workload 3, where only a handful of the initial artifacts are materialized. Therefore, HL has to re-execute many of the operations at the end of Workload 3, which results in an overhead of around 70 seconds when compared to CO.

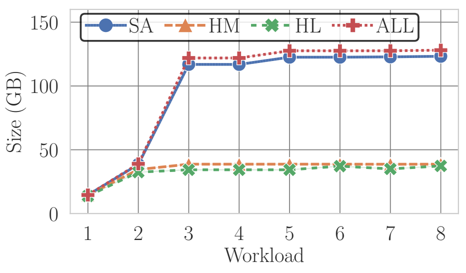
Figure 2.5 shows the cumulative run time of executing the Kaggle workloads consecutively. Workloads 4-8 operate on the artifacts generated in Workloads 1-3; thus, instead of recomputing



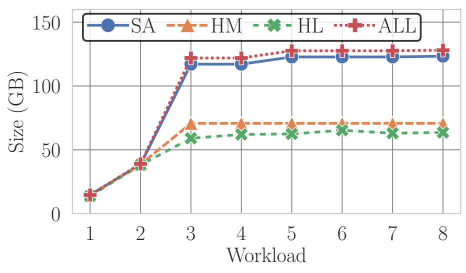
(a) Budget = 8 GB.



(b) Budget = 16 GB.



(c) Budget = 32 GB.



(d) Budget = 64 GB.

Figure 2.6: Real size of the materialized artifact for different materialization strategies.

those artifacts, CO reuses the artifacts. As a result, the cumulative run time of running the 8 workloads decreases by 50%. HL also improves run time when compared to KG. However, HL only materializes the initial artifacts of the workloads and has a smaller improvement over KG when compared to CO.

This experiment shows that optimizing a single execution of each workload improves the run time. In a real collaborative environment, there are hundreds of modified scripts and possibly thousands of repeated execution of such scripts, resulting in 1000s of hours of improvement in run time.

2.8.3 Materialization

In this set of experiments, we investigate the impact of different materialization algorithms on storage and run time.

Effect of Materialization on Storage. In a real collaborative environment, deciding on a reasonable materialization budget requires knowledge of the expected size of the artifacts, the number of users, and the rate of incoming workloads. In this experiment, we show that even with a small budget, our materialization algorithms, particularly our storage-aware algorithm, store a large portion of the artifacts that reappear in future workloads. We hypothesize that there is considerable overlap between columns of different datasets in ML workloads. Therefore, the actual total size of the artifacts that our storage-aware algorithm materializes is larger than the specified budget.

We run the Kaggle workloads under different materialization budgets and strategies. Figure 2.6 shows the real size of the stored artifacts for the heuristics-based (HM), storage-aware (SA), and Helix (HL) algorithms. To show the total size of the materialized artifacts, we also implement a strategy that materializes every artifact in EG (represented by ALL in the figure). In HM, the maximum real size is always equal to the budget. This is similar for HL since it does not perform any compression or deduplication of the columns. However, in SA, the real

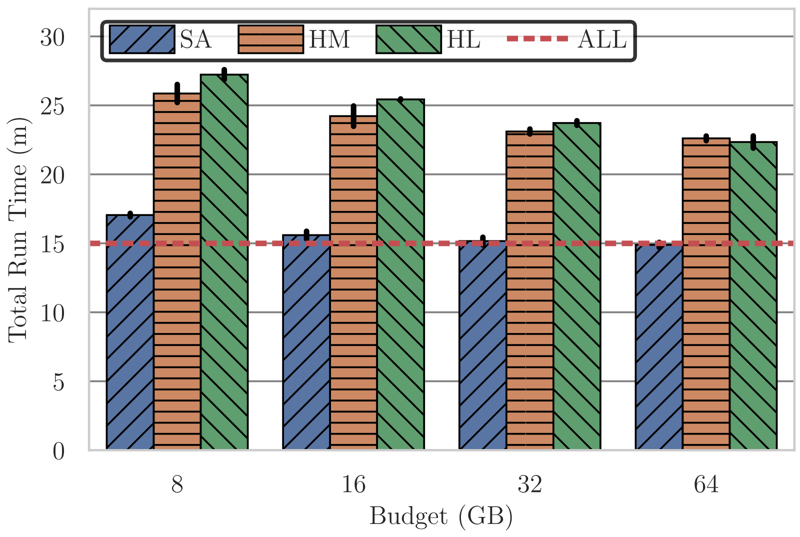


Figure 2.7: Total run time of the Kaggle workloads with different budgets and strategies.

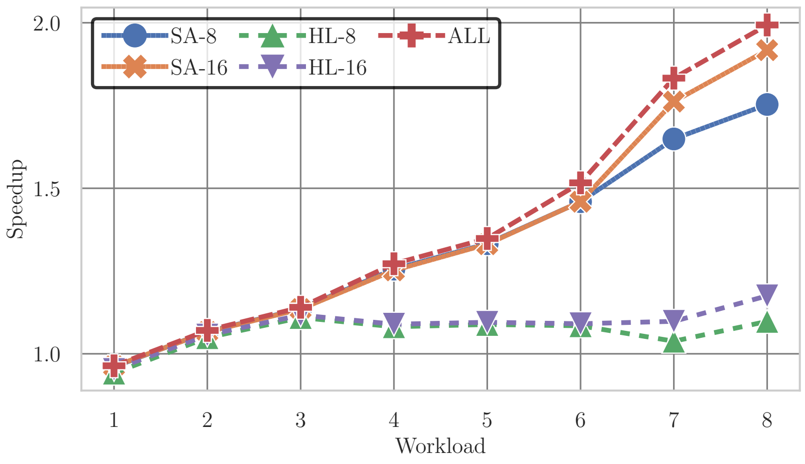


Figure 2.8: Speed up of different materialization strategies and budgets over the baseline.

size of the stored artifacts reaches up to 8 times the budget. With a materialization budget of 8 GB and 16 GB, SA materializes nearly 50% and 80% of all the artifacts. For budgets larger than 16 GB, SA materializes nearly all of the artifacts. This indicates that there is considerable overlap between the artifacts of ML workloads. By deduplicating the artifacts, SA can materialize more artifacts.

Note that when a high-utility artifact has no overlap with other artifacts, SA still prioritizes it over other artifacts. As a result, it is likely that when materializing an artifact that has no overlap with other artifacts, the total size of the materialized data decreases. Figure 2.6a shows such an example. After executing Workload 2, SA materializes several artifacts that overlap with each other. However, in Workload 3, SA materializes a new artifact with a high utility, which represents a large dataset with many features (i.e., 1133 columns and around 3 GB). Since the new artifact is large, SA removes many of the existing artifacts. As a result, the total size of the materialized artifacts decreases after Workload 3.

Effect of Materialization on Run time. Figure 2.7 shows the total run time of different materialization algorithms and budgets. ALL represents the scenario where all the artifacts are

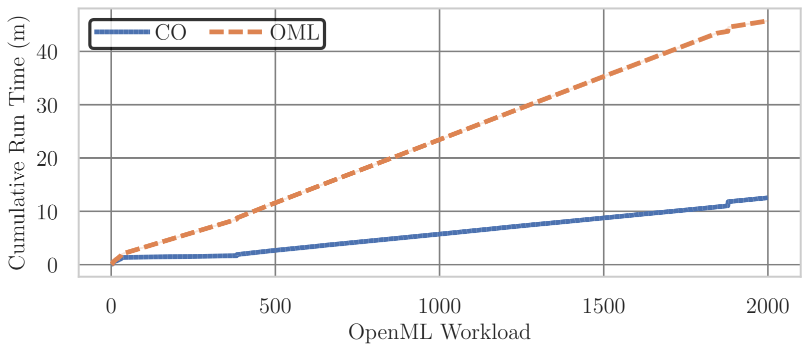


Figure 2.9: Impact of quality-based materialization on the run time.

materialized. Even with a materialization budget of 8 GB, SA has comparable performance to ALL (i.e., a difference of 100 seconds in run time). When the budget is larger than 8 GB, SA performs similarly to ALL. For small materialization budgets ($\leq$ 16 GB), HM performs 50% worse than SA. However, HM performs slightly better for larger materialization budgets. The difference between HM and SA is because many of the artifacts are large, e.g., in Workload 3, some artifacts are more than 3 GB. Most of these artifacts contain overlapping columns and SA compresses them. However, HM is unable to exploit this similarity and chooses not to materialize any of the large artifacts. Recomputing these artifacts is costly, which results in a larger run time for HM.

HL does not prioritize the artifacts based on their cost or potential. Thus, HL quickly exhausts the budget by materializing initial artifacts. The impact of such behavior is more visible for smaller budgets ($\leq$ 16 GB), where HL performs 20% and 90% worse than HM and SA, respectively. For larger budgets, HL has a similar performance to HM, since a larger fraction of all the artifacts is materialized.

In Figure 2.8, we plot the cumulative speedup (vs the KG baseline) of different materialization algorithms and budgets. We plot the speedup of SA and HL with budgets of 8 GB and 16 GB (SA-8, SA-16, HL-8, and HL-16 in the figure) as the rest of the algorithms and budgets show similar behavior. ALL achieves a speedup of 2 after executing all the workloads. SA has a comparable speedup with ALL reaching speedups of 1.77 and 1.97 with budgets of 8 GB and 16 GB, respectively. Since HL only materializes the initial artifacts, it only provides a small speedup over the KG baseline. After executing all the workloads, HL reaches speedups of 1.11 and 1.18 with budgets of 8 GB and 16 GB. For larger budgets (i.e., 32 GB and 64 GB), HL reaches a maximum speedup of 1.31. Whereas, SA has a speedup of 2.0, similar to ALL.

Effect of Model Quality on Materialization. In many collaborative ML use cases, users tend to utilize existing high-quality models. In our materialization algorithm, we consider model quality when materializing an artifact. In this experiment, we show the impact of materializing high-quality models on run time and show that our materialization algorithm quickly detects high-quality models.

We design a model-benchmarking scenario, where users compare the score of their models with the score of the best-performing model found so far. Such a scenario is common in collaborative settings, where users constantly attempt to improve the best current model. We use the OpenML workloads for the model-benchmarking scenario. The implementation of the

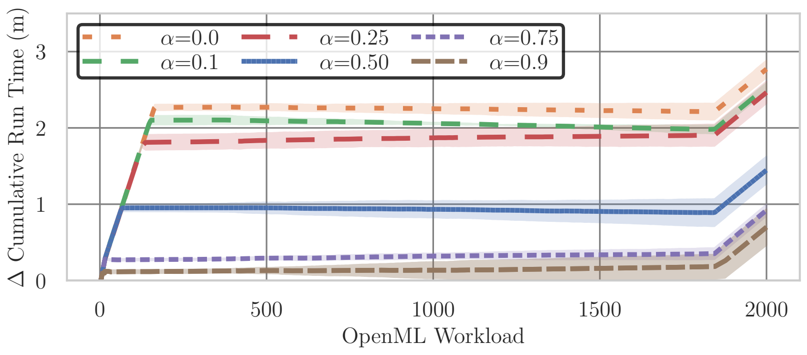


Figure 2.10: Impact of α on the run time. The figure shows the difference in run time to when $\alpha = 1$ (i.e., the gold standard model is always materialized).

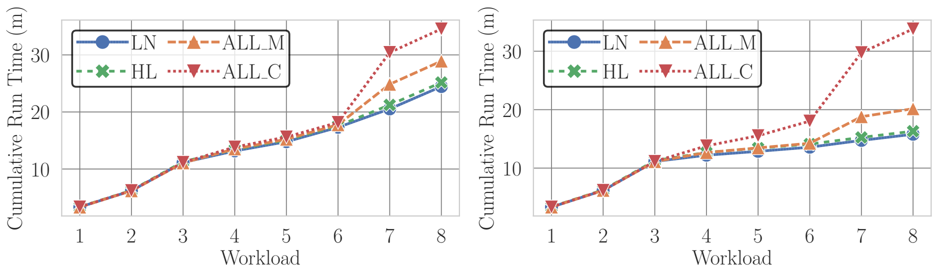
scenario is as follows. First, we execute the OpenML workloads one by one and keep track of the workload with the best performing model, which we refer to as the *gold standard workload*. Then, we compare every new workload with the gold standard.

Figure 2.9 shows the cumulative run time of the model-benchmarking scenario using our collaborative optimizer (CO) with default configuration (i.e., storage-aware materializer with budget 100 MB and $\alpha = 0.5$) against the OpenML baseline (OML). For every new workload, OML has to re-run the gold standard workload. When CO encounters a gold standard workload, the materialization algorithm assigns a higher potential value to the artifacts of the workload. As a result, such artifacts have a higher materialization likelihood. In the subsequent workloads, CO reuses the materialized artifacts of the gold standard from EG instead of re-running them, resulting in a 5 times improvement in the run time over OML. Re-executing the gold standard workload results in an overhead of 2000 seconds, which contributes to the large run time of OML. In comparison, reusing the artifacts of the gold standard has an overhead of 65 seconds for CO.

We also investigate the impact of α, which controls the importance of model quality in our materialization, on the run time of the model-benchmarking scenario. If α is close to 1, the materializer aggressively stores high-quality models. If α is close to 0, the materializer prioritizes the recreation time and size over quality. The materialization budget for the OpenML workloads is 100 MB. However, the models in OpenML are typically small (less than 100 KB). Therefore, regardless of the α value, the materializer stores the majority of the artifacts, which makes it difficult to accurately study the effect of the α value. Therefore, in this experiment, we set the budget to one artifact (i.e., the materializer is only allowed to store one artifact). An ideal materializer always selects the gold standard model. This highlights the impact of α on materialization more clear.

We run the model-benchmarking scenario and vary the value of α from 0 to 1. When α is 1, the materializer always materializes the gold standard model, as it only considers model quality. Therefore, $\alpha = 1$ incurs the smallest cumulative run time in the model-benchmarking scenario.

In Figure 2.10, we report the difference in cumulative run time between $\alpha = 1$ and other values of α (i.e., y-axis corresponds to the delta in cumulative run time when compared to $\alpha = 1$). In the scenario, we repeatedly execute the gold standard; thus, the faster we materialize the gold standard model, the smaller the cumulative run time would become. Once



(a) Heuristics-based materialization. **(b)** Storage-aware materialization.

Figure 2.11: Run time of the Kaggle workloads for different reuse strategies for both heuristics-based and storage-aware materialization.

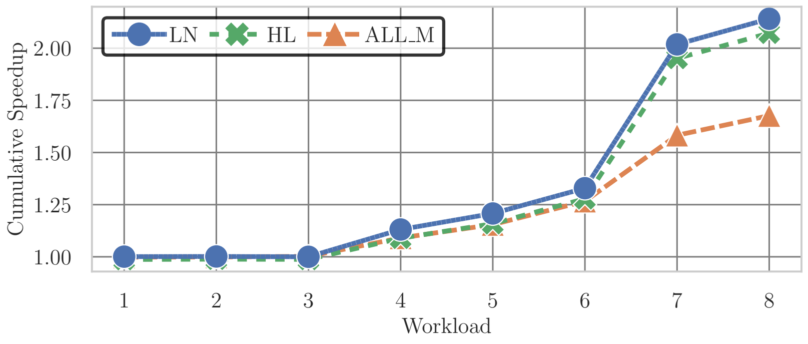


Figure 2.12: Speed up of different reuse strategy over the naive all compute approach.

we materialize the gold standard model, the delta in cumulative run time reaches a plateau. This is because the overhead of reusing the gold standard is negligible; thus, cumulative run time becomes similar to when $\alpha = 1$. In workload 14, we encounter a gold standard that remains the best model until nearly the end of the experiment. Smaller α values ($\alpha \leq 0.25$) materialize this model after more than 100 executions. As a result, their delta in run time reaches a plateau later than large α values ($\alpha \geq 0.5$). The long delay in the materialization of the gold standard contributes to the higher cumulative run time for smaller values of α.

The default value of α in our system is 0.5. This value provides a good balance between workloads that have the goal of training high-quality models (e.g., the model-benchmarking scenario) and workloads that are more exploratory in nature. When we have prior knowledge of the nature of the workloads, then we can set α accordingly. We recommend $\alpha > 0.5$ for workloads with the goal of training high-quality models and $\alpha < 0.5$ for workloads with exploratory data analysis.

2.8.4 Reuse

In this experiment, we compare our linear-time reuse algorithm (LN) with Helix (HL) and two other baselines (ALL_M and ALL_C). ALL_M reuses every materialized artifact. ALL_C recomputes every artifact (i.e., no reuse).

Figures 2.11a and 2.11b show the run time of the Kaggle workloads with different materialization algorithms. ALL_C, independent of the materialization algorithm, finishes the execution of the workloads in around 2000 seconds. For the heuristics-based materialization, all four reuse algorithms have similar performance until Workload 6. This is because Workload 3

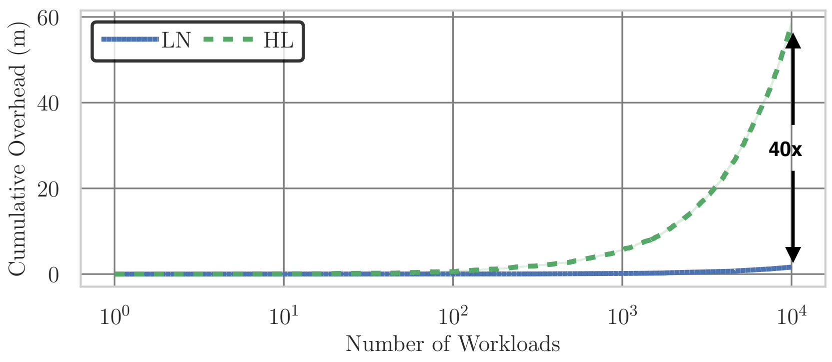


Figure 2.13: Overhead of the linear-time reuse algorithm of our collaborative workload optimizer vs the reuse algorithm of Helix.

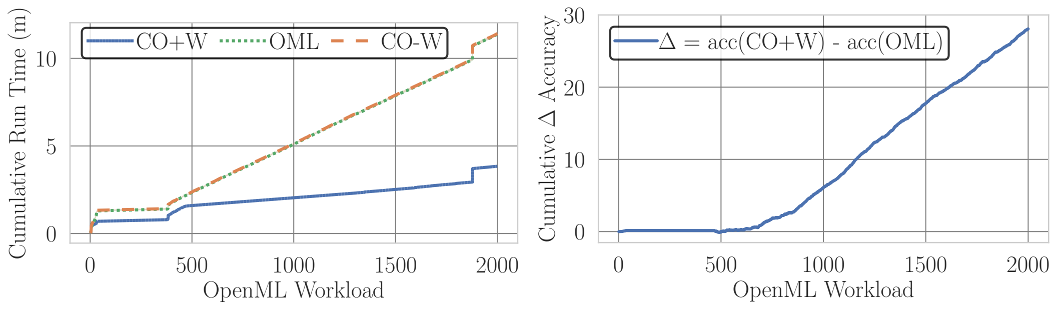
has large artifacts and the heuristics-based materialization exhausts its budget by materializing them. Furthermore, Workloads 4, 5, and 6 are modified versions of Workloads 1 and 2 (Table 2.1). As a result, there are not many reuse opportunities until Workload 7, which is a modified version of Workload 3.

The storage-aware materialization (Figure 2.11b) has better budget utilization and materializes some of the artifacts of Workloads 1 and 2. ALL_M, LN, and HL reuse these artifacts in Workloads 4, 5, and 6; thus, improving the run time from Workload 4. To better show the impact of the reuse algorithms, we plot the cumulative speedup of LN, HL, and ALL_M over ALL_C for the storage-aware materialization in Figure 2.12. Since the first three workloads do not share many similar artifacts, the speedup is 1. However, after the third workload, all the reuse algorithms have speedups larger than 1. After executing all workloads, LN and HL reach a speedup of around 2.1 with LN slightly outperforming HL.

For both materialization strategies, ALL_M has a similar performance to LN and HL until Workload 6. Many of the artifacts of Workload 7 incur larger load costs than compute costs. As a result, LN and HL recompute these artifacts and result in a smaller cumulative run time than ALL_M, i.e., around 200-300 seconds. In this experiment, since EG is inside the memory of the machine, load times are generally low. LN and HL outperform ALL_M with a larger margin in scenarios where EG is on disk.

Reuse Overhead. The polynomial-time reuse algorithm of Helix generates the same plan as our linear-time reuse algorithm. For the Kaggle workloads, since the number and the size of workloads are relatively small, we only observe a small difference of 5 seconds in the reuse overhead.

To show the impact of our linear-time reuse algorithm, we perform an experiment with 10,000 synthetic workloads. We design the synthetic workloads to have similar characteristics to the real workloads in Table 2.1. We consider the following 5 attributes of the real workload DAGs: (1) indegree distribution (i.e., join and concat operators), (2) outdegree distribution, (3) ratio of the materialized nodes, (4) distribution of the compute costs, and (5) distribution of the load costs. A node with an outdegree of more than 1 represents a scenario where the node is input to different operations (e.g., training different ML models on one dataset node). To generate the workloads, we first randomly select the number of nodes inside the workload DAG from the [500, 2000] interval, which represents many of the Kaggle workloads. Then, for every node, we sample its attributes from the distributions of attributes of the real workloads.



(a) Run time. **(b)** Difference in accuracy.

Figure 2.14: Impact of Warmstarting on the run time and accuracy for the OpenML workloads.

Figure 2.13 shows the cumulative overhead of LN and HL on 10,000 generated workloads. The overhead of LN increases linearly and after the 10,000s workloads, LN incurs a total overhead of 80 seconds. In comparison, HL has an overhead of 3500 seconds, 40 times more than LN. In a real collaborative setting, where hundreds of users are executing workloads, a large reuse overhead leads to slower response time and may cause a bottleneck during the optimization.

2.8.5 Warmstarting

In this experiment, we evaluate the effect of our warmstarting method. Figure 2.14 shows the effect of warmstarting on run time and accuracy for the OpenML workloads. In Figure 2.14a, we observe that the cumulative run time of the baseline (OML) and our optimizer without warmstarting (CO-W) are similar. In the OpenML workloads, because of the small size of the datasets, the run time of the data transformations is only a few milliseconds. As a result, CO-W only improves the average run time by 5 milliseconds when compared to OML. The model training operations are the main contributors to the total run time. Warmstarting the training operations has a large impact on run time. As a result, warmstarting (CO+W) improves the total run time by a factor of 3.

In Figure 2.14b, we show the cumulative difference between the accuracy (Δ Accuracy) of the workloads with and without warmstarting. For example, for a hundred workloads, if warmstarting improves the accuracy of each workload by 0.02, then the cumulative Δ accuracy is 2.0. Figure 2.14b shows that warmstarting can also lead to an improvement in model accuracy. This is mainly due to the configuration of the OpenML workloads. Apart from the convergence criteria (i.e., model parameters do not change), most of the workloads in OpenML have termination criteria as well. For example, in the logistic regression model, users set the maximum number of iterations. In such cases, warmstarting can improve model accuracy since training starts from a point closer to the convergence. As a result, warmstarting finds a better solution when reaching the maximum number of iterations. For the OpenML workloads, warmstarting leads to an average Δ accuracy of 0.014.

2.9 Conclusion

In this chapter, we present our solution for optimizing collaborative ML workloads. We propose a graph representation of the artifacts of ML workloads, which we refer to as Experiment Graph (EG). Using EG, we offer materialization and reuse algorithms. We propose two materialization algorithms. The heuristics-based algorithm stores artifacts based on their likelihood of reappearing in future workloads. The storage-aware algorithm takes deduplication information of the artifacts into account when materializing them. Given the set of materialized artifacts, for a new workload, our reuse algorithm finds the optimal execution plan in linear time.

To evaluate our optimization techniques, we implement a system prototype and utilize real-world workloads from Kaggle [26] and OpenML [27]. We show that our solution improves the execution time of collaborative workloads by more than one order of magnitude for repeated executions and by 50% for modified workloads. We also show that our storage-aware materialization can store up to 8 times more artifacts than the heuristics-based materialization algorithm. Our reuse algorithm finds the optimal execution plan in linear time and outperforms the state-of-the-art polynomial-time reuse algorithm.

The collaborative setting, which we introduce in this chapter, is the first phase of the life-cycle of the machine learning applications. The result of the collaborative workloads is one or a set of candidate pipelines. To fully utilize the pipelines, data scientists deploy the candidate pipelines into production environments for answering prediction queries. In the following chapters, we introduce two settings for the deployment and continuous training of ML pipelines. We also propose several optimization techniques to improve the execution performance of ML workloads.

3
Single-pipeline Deployment and Continuous Training

In many ML applications, the result of the collaborative workloads is a pipeline. To utilize this pipeline for answering prediction queries, data scientists deploy this pipeline into a deployment environment. In this chapter, we introduce the single-pipeline deployment setting and workloads. After the initial deployment, 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 accurate predictions, 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 propose online statistics computation and dynamic materialization of the preprocessed features, which further reduces the total training and data preprocessing time. To evaluate our proposed optimizations, we implement a system prototype. Furthermore, we design and deploy two pipelines and models to process two real-world datasets. Our experiments show that the proposed optimization techniques reduce the total training cost up to 15 times for single-pipeline deployment workloads. Furthermore, our approach provides the same level of model quality when compared to the state-of-the-art deployment approaches.

3.1 Motivation

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 [17]. 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, online learning is sensitive to noise and outliers, which may increase the prediction error rate. Therefore, to guarantee a high level of quality, one has to tune the online learning method to the specific use case [20, 50]. Thus, effective online deployment of machine learning models cannot provide robustness and simplicity.

To solve the problem of degrading model quality, a 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 [51]. 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.

3.2 Research Contributions

We propose a solution for optimizing the single-pipeline deployment workloads. Our solution comprises 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 approach, the tuning process of our deployment platform is simple and requires the same amount of user interaction as the periodical deployment.

In our deployment platform, we continuously update 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 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, update 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 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, we make the following contributions.

- We design 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.
- We offer proactive training of the deployed models and pipelines that frequently updates the model using samples of the data. Proactive training completely eliminates the need for periodical retraining while providing the same level of model quality.
- We propose efficient data processing and model training by utilizing online statistics computation and dynamic materialization, which provide up-to-date models for answering prediction queries.

The rest of this chapter is organized as follows. In Section 3.3, we provide background information on the training strategy we utilize in our continuous deployment platform and the tuning mechanism of the existing deployment platforms. Section 3.4 describes the details of our continuous training approach. This section contains the details of our optimization techniques, i.e., online statistics computation, dynamic materialization, and proactive training. In Section 3.5, we introduce the architecture of our deployment platform. In Section 3.6, we evaluate the performance of our continuous deployment platform. Finally, Section 3.7 presents our conclusion.

3.3 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) [52]. 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.

3.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a function optimization strategy utilized by many machine learning algorithms for training a model. SGD is an iterative optimization technique. In every iteration, SGD utilizes one data point or a sample of the data points to update the model. SGD is suitable for large datasets as it does not require scanning the entire data in every iteration [25]. 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 [52, 50], clustering [53], and matrix factorization [54] utilize SGD in training models. SGD is also the most common optimization strategy for training neural networks on large datasets [55].

To explain the details of SGD, we describe how it is utilized to train a simple linear regression model. In linear regression, the goal is to find the weight vector (w) that minimizes the least-squares cost function ($J(w)$):

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}(x^i w - y^i)^2 \tag{3.1}$$

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:

$$w^{t+1} = w^t + \eta \sum_{i \in S}(y^i - x^i w)x^i \tag{3.2}$$

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 gradually decrease it after every iteration. However, in complex and high-dimensional problems, the simple tuning approach is ineffective [56]. Adaptive learning rate methods such as Momentum [57], Adam [58], Rmsprop [59], and AdaDelta [60] 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 [58, 59, 60]. 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 a 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.

3.3.2 Tuning the Periodical Deployment

Typically, two groups of hyperparameters affect the efficiency of the periodical deployment approach. The first group (the deployment hyperparameters) controls the frequency and amount of data for every retraining. The second group (the training hyperparameters) tunes the algorithm for the 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 [61, 62]. 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 (hyperparameter for the amount of data for every retraining) on a daily basis (hyperparameter for the frequency of the retraining). In the next sections, we describe how we tune the deployment and training hyperparameters of our deployment framework.

3.4 Continuous Training Approach

Our solution for optimizing the execution of workloads in the single-pipeline deployment setting is a platform that manages and continuously trains the deployed pipeline. At the core of the platform are our three proposed techniques, namely, online statistics computation, dynamic materialization, and proactive training. In this section, we describe the details of our solution

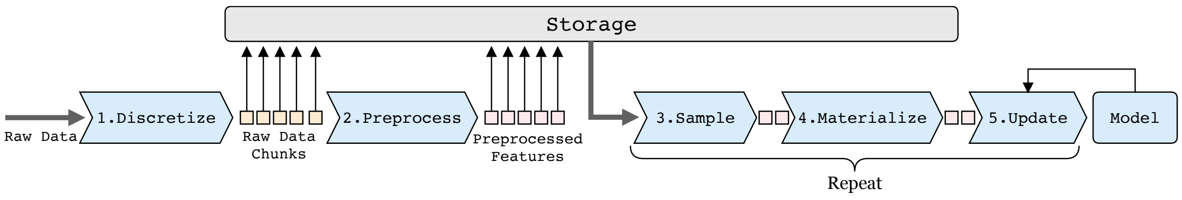


Figure 3.1: Workflow of our continuous deployment approach. (1) The platform converts the data into small units. (2) The platform utilizes the deployed pipeline to preprocess the data and transform the raw data into features and store them in the storage. (3) The platform samples the data from the storage. (4) The platform materializes the sampled data. (5) Using the sampled data, the deployment platform updates the deployed model.

and present our proposed platform. Figure 3.1 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. Each generated feature chunk contains a reference to the originating raw data chunk. The platform stores the feature chunks in the storage unit. When the storage unit becomes full, the platform removes the oldest feature chunks but keeps the reference to the raw data chunks. If 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. 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.

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*.

In the rest of this section, we first describe the details of the online statistics computation during the preprocessing steps. Then, we introduce the dynamic materialization approach and the effect of different sampling strategies on the materialization process. Finally, we describe the details of the proactive training method.

3.4.1 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 learning 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 internal statistics. Finally, the component transforms and forwards the data to the next component. Online statistics computation eliminates the need to recompute the statistics during dynamic materialization and proactive training.

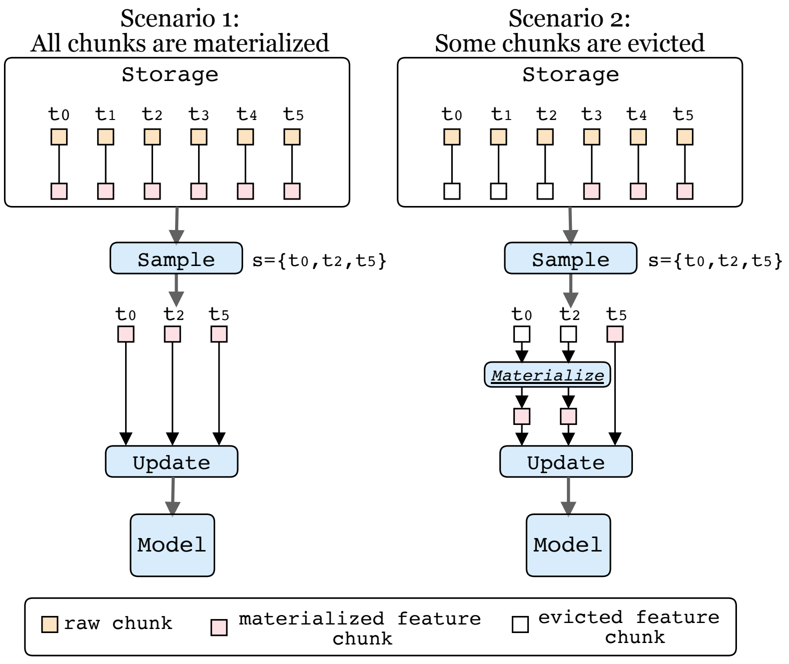
Online statistics computation is only applicable to certain types of pipeline components. The support for stateless pipeline components is trivial as they do not rely on any statistics before transforming the data. For stateful operations, since the statistics update occurs during the online data processing, the platform can only update the statistics that can be computed incrementally. Many well-known data preprocessing components, such as standardization and one-hot encoding, can compute their statistics incrementally (e.g., mean, standard deviation, and a hash table). However, some pipeline components require statistics that are not incrementally computed (such as percentile). Furthermore, some components utilize non-incremental algorithms for computing models (e.g., PCA). As a result, our deployment platform does not support such components. Fortunately, recent bodies of work have developed novel techniques for online feature engineering [63, 64] and approximate machine learning [65] that offer fast and incremental alternatives with theoretical error bounds to non-incremental algorithms.

The platform can also facilitate the online statistics computation for user-defined pipeline components. In Section 3.5, we describe how users can incorporate this feature into their custom pipeline components.

3.4.2 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 we can skip the entire data preprocessing steps when updating the model. However, in the 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, the platform removes the content of the oldest materialized feature chunks when the size of the stored feature chunks exceeds the storage capacity (similar to cache eviction). However, the platform keeps the unique identifier and the reference to the raw data chunk. The 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 3.2 shows the process of dynamic materialization in two possible scenarios. For both scenarios, there are a total of 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



Scenario 1: All chunks are materialized
Scenario 2: Some chunks are evicted
Storage
t0 t1 t2 t3 t4 t5
Sample
s={t0,t2,t5}
Materialize
Update
Model
raw chunk
materialized feature chunk
evicted feature chunk

Figure 3.2: The process of dynamic materialization.

Table 3.1: Description of the pipeline component types. Unit of work indicates whether the component operates on a row or a column.

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

model. In Scenario 2, the platform has previously evicted some of the materialized feature chunks due to the limited storage capacity. In this scenario, the platform first re-materializes the evicted chunks using the deployed pipeline components before updating the model.

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.

3.4.3 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 3.1 shows different pipeline component categories and their characteristics. Let us assume the total number of the values in a dataset where $\mathcal{R}$ represents the rows and $\mathcal{C}$ represents columns is p, where $p = |\mathcal{R}| \times |\mathcal{C}|$. Data transformation and feature selection operations either perform a one-to-one mapping (e.g., normalization) or remove some rows/columns (e.g., anomaly filtering and variance thresholding). Therefore, the complexity of data transformation and feature selection operations is linear in terms of the input size ($\mathcal{O}(p)$). 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 $\mathcal{O}(p)$ as the increase in size is linear with respect to the input size. However, in some cases, 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 $\forall c \in \mathcal{C}$. Furthermore, let us assume $q = \max_{\forall c \in \mathcal{C}} |\mathcal{U}(c)|$, 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 $\mathcal{O}(pq)$ (each existing value is encoded with at most q binary values). Based on the value of q, two scenarios may occur:

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

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 $\mathcal{O}(p^2)$). 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 $\mathcal{O}(p)$.

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

3.4.4 Effects of Sampling Strategies on Dynamic Materialization

Our platform offers three sampling strategies, namely, uniform, window-based, and time-based (Section 3.5.2). The choice of the sampling strategy affects the efficiency of 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

[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

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 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:

$$\mu = \frac{\sum_{n=1}^{N} \mu_n}{N} \tag{3.3}$$

μ indicates the ratio of the feature chunks that do not require re-materialization (a μ 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 3.5.1). Next, we describe how the sampling strategy affects the computation of μ.

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

$$\mu_n = \begin{cases} 1, & \text{if } n \leq m \\ \dfrac{E_n[MS]}{s} = \dfrac{s\frac{m}{n}}{s} = \dfrac{m}{n}, & \text{otherwise} \end{cases}$$

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

$$\begin{aligned} \mu = \frac{\sum\limits_{n=1}^{N} \mu_n}{N} &= \frac{m \times 1.0 + \sum\limits_{n=m+1}^{N} \frac{m}{n}}{N} \\ &= \frac{m + m(\frac{1}{m+1} + \frac{1}{m+2} + ... + \frac{1}{N})}{N} \\ &= \frac{m(1 + (H_N - H_m))}{N} \\ &\approx \frac{m(1 + ln(N) - ln(m))}{N} \end{aligned} \tag{3.4}$$

The highlighted section corresponds to the Harmonic numbers [66]. The t-th harmonic number is:

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

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 $m \geq w$ then $\mu = 1$, as all the feature chunks in the active window are always materialized. However, when $m < w$:

$$\mu_n = \begin{cases} 1, & \text{if } n \leq m \\ \dfrac{E_n[MS]}{s} = \dfrac{m}{n}, & \text{if } m < n \leq w \\ \dfrac{E_w[MS]}{s} = \dfrac{m}{w}, & \text{if } w < n \end{cases}$$

therefore:

$$\begin{aligned} \mu = \frac{\sum_{n=1}^{N} \mu_n}{N} &= \frac{m + \sum_{n=m+1}^{w} \frac{m}{n} + (N-w)\frac{m}{w}}{N} \\ &\approx \frac{m + m(H_w - H_m) + (N-w)\frac{m}{w}}{N} \\ &= \frac{m(1 + ln(w) - ln(m) + \frac{N-w}{w})}{N} \end{aligned} \tag{3.5}$$

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 3.4, 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 any data 91% of the time.

3.4.5 Proactive Training

Updating the model is the last step of our continuous deployment platform. We update the model through the proactive training process. The full retraining process is triggered by a certain event, such as a drop in the quality or a certain amount of time elapsed since the last retraining. However, proactive training continuously updates the deployed model. 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 mini-batch SGD. Algorithm 3 shows the pseudocode of the mini-batch SGD algorithm.

In mini-batch SGD, we first initialize the model (Line 1). Then, in every iteration, we randomly sample points from the dataset (Line 3), compute the gradient of the loss function J (Line 4), and finally update the model based on the value of the gradient and the learning rate (Line 5). 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

Algorithm 3: Mini-batch stochastic gradient descent.

```
  Input: $D$ = training dataset
  Output: $m$ = trained model
1 initialize $m_0$;
2 for $i = 1...n$ 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 return $m_n$;
```

these parameters, iterations of SGD are conditionally independent 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 [67], which is limited to SGD-based models. In our deployment platform, one can replace 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 variety of machine learning models in different domains [50, 53, 54]. 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 [52, 58].

3.5 Deployment Platform

Our proposed deployment platform comprises five main components: pipeline manager, data manager, scheduler, proactive trainer, and execution engine. Figure 3.3 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.

3.5.1 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

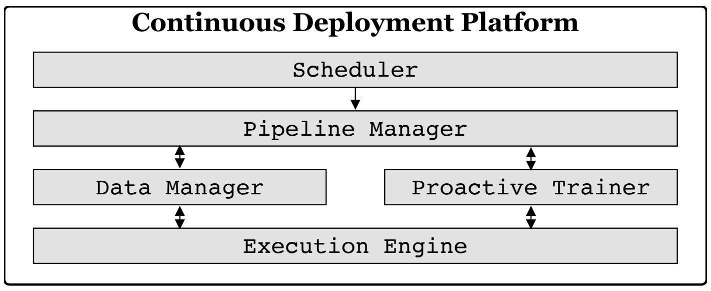


Figure 3.3: Architecture of the continuous deployment platform.

proactive training. The scheduler uses the following formula to compute the time when to execute the next proactive training:

$$T' = S * T * pr * pl \tag{3.6}$$

where T' is the time in seconds when the next proactive training is scheduled to execute, T is the total execution time (in seconds) of the last 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 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.

3.5.2 Data Manager

The data manager component is responsible for the storage of the historical data and the 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 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 a key. Then, the data manager proceeds as follows. For every sampled timestamp, if the transformed feature chunk exists in the storage, 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. In time-based sampling, recent chunks have a higher probability of being sampled. The window-based sampling strategy is similar to uniform sampling, but instead of sampling from the entire historical data, the data manager samples the data from a given time window. 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., classification of images). In these scenarios, the window-based and the time-based sampling strategies may fail to provide a non-biased sample. In Section 3.6, we evaluate the effect of the sampling strategy on both the total deployment cost and the quality of the deployed model.

3.5.3 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 a sample of the data chunks. 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 [18]. Finally, the pipeline manager utilizes the deployed model to make predictions.

3.5.4 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 (a sample of the materialized feature chunks) 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 such as Adam, Rmsprop, and AdaDelta to dynamically adjust the learning rate parameter when training the model.

In order for 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 3.4.5, the proactive training with data sampling can guarantee convergence only when the SGD optimization is utilized.

3.5.5 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 [38], Apache Flink [68], and GoogleDataFlow [69] are distributed data processing platforms that support both stream and batch data processing.

3.6 Evaluation

To evaluate the impact of our optimization techniques and 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.

Table 3.2: Description of the 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).

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

3.6.1 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 [70]. 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 [71]. 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 Spark RDDs. Therefore, we can utilize the caching mechanism of Apache Spark to materialize and un-materialize feature chunks.

Datasets. Table 3.2 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

[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)`

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.

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 [72]. 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).

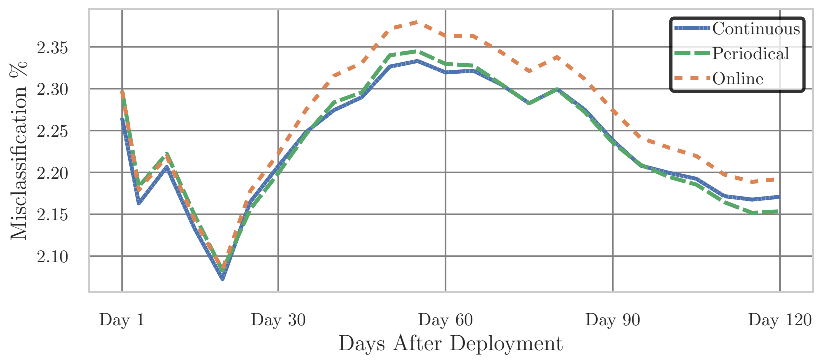
3.6.2 Experiment 1: Deployment Approaches

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.

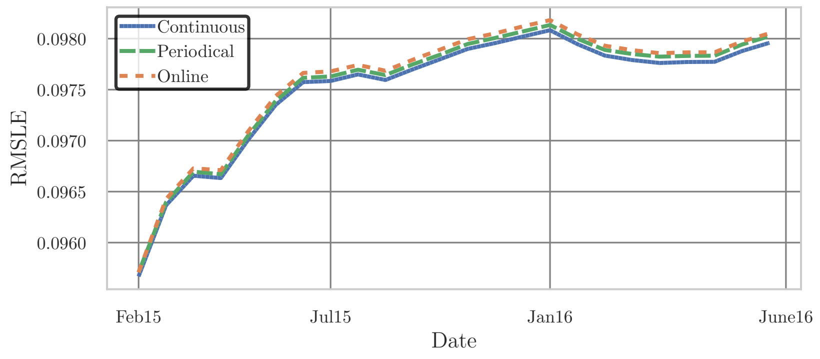
- Online: deploy the pipeline, then utilize online gradient descent with Adam learning rate adaptation method for updating the deployed model.
- Periodical: deploy the pipeline, then periodically retrain the deployed model.
- Continuous: deploy the pipeline, then continuously update the deployed model 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 warmstarting technique, used in the TFX framework [18]. In warmstarting, 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 3.4 shows 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 3.4a, during the first 110 days of the deployment, the continuous deployment has a lower error rate than the periodical



(a) Misclassification error rate for the URL dataset.



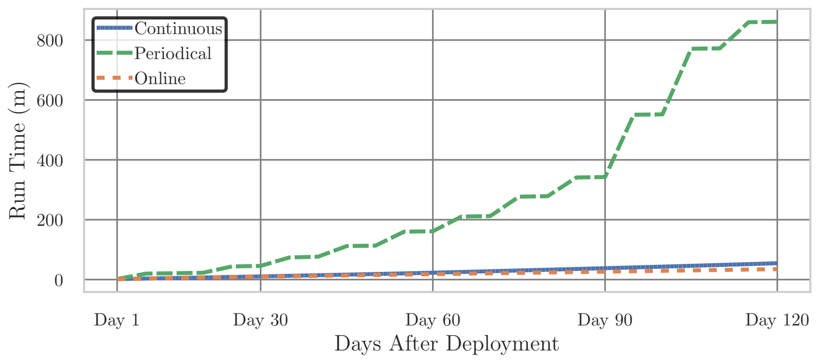
(b) Root mean squared logarithmic error rate for the Taxi dataset.

Figure 3.4: Model quality for the URL and Taxi datasets after deployment.

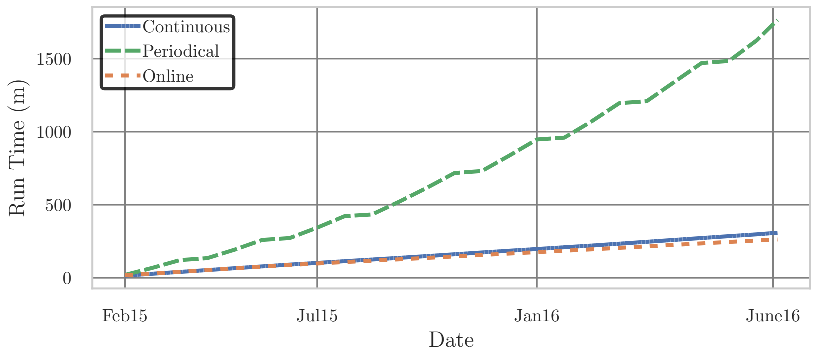
deployment. Only after the final retraining, the periodical deployment slightly outperforms the continuous deployment. However, from the start to the end of the deployment process, the continuous deployment improves the average error rate by 0.3% and 1.5% over the periodical and online deployment, respectively. In Figure 3.4b, 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 3.5, 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 predictions. For the URL dataset (Figure 3.5a), 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.



(a) URL dataset.



(b) Taxi dataset.

Figure 3.5: Cumulative run time (data processing + model training) for the URL and Taxi datasets after deployment.

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 3.5b), 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 the URL dataset).

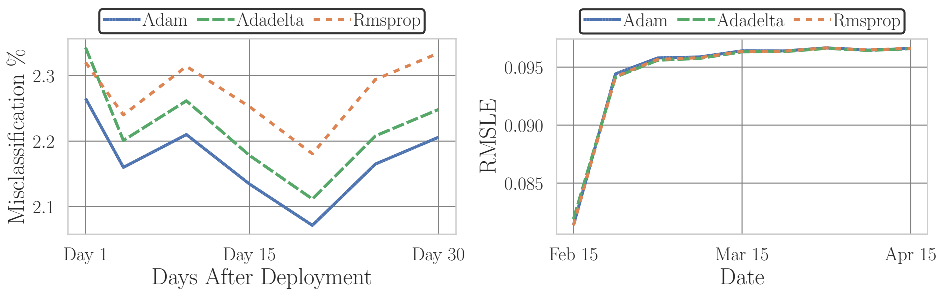
3.6.3 Experiment 2: System Tuning

In this experiment, we investigate the effect of different parameters on the quality of the models after deployment. As described in Section 3.4.5, 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 to also perform 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

Table 3.3: Hyperparameter tuning during initial training (bold numbers show the best results for each adaptation techniques).

	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



(a) URL dataset.

(b) Taxi dataset.

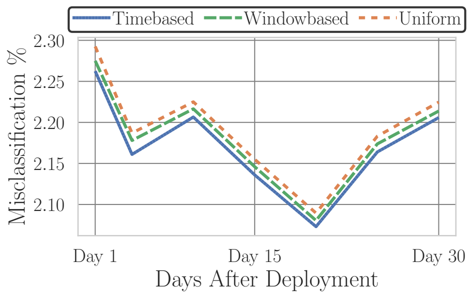
Figure 3.6: Result of hyperparameter tuning during the deployment.

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 3.2) 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 3.3 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 hyperparameters 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.

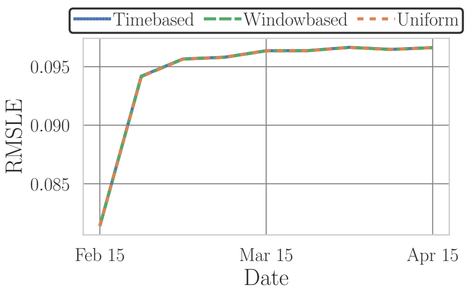
After the initial training, for every configuration, we deploy the model and use 10 % of the remaining data to evaluate the model after deployment. Figure 3.6 shows the results of the different hyperparameter configurations on the deployed model. To make the deployment figure more readable, we avoid 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.



(a) URL dataset.



(b) Taxi dataset.

Figure 3.7: Effect of different sampling methods on quality.

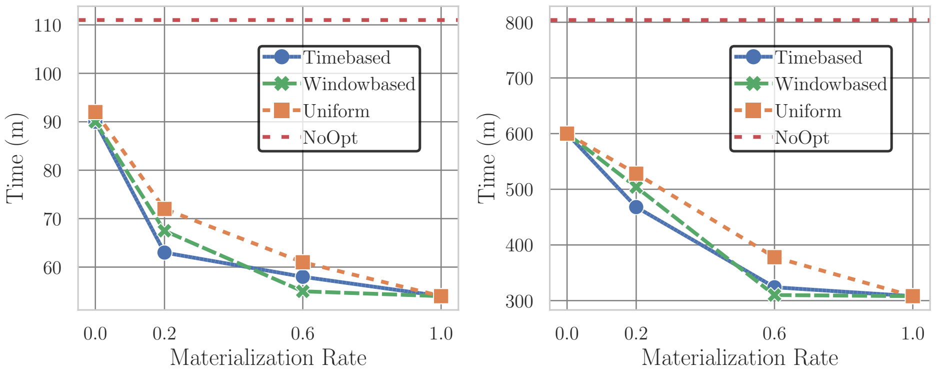
Therefore, the quality of the model after an update is directly related to the quality of the sample. We evaluate the effect of three different sampling strategies, namely, time-based, window-based, and uniform, on the quality of the deployed model. The sample size is similar to the sample size during the initial training ($16k$ and $1M$ data points for URL and Taxi datasets, respectively). Figure 3.7 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 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 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.

3.6.4 Experiment 3: Optimizations Effects

In this experiment, we analyze the effect of the system optimizations, i.e., online statistics computation and dynamic materialization on the total deployment cost. We define the materialization rate (i.e., $\frac{m}{n}$, as described in Section 3.4.2) 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). In this experiment, we assume the raw data is always stored in memory. The total size of the datasets after materialization is 5.2 GB and 59 GB for the URL and Taxi datasets, respectively. Therefore, when setting the materialization rate to a specific value, we must ensure we have enough memory capacity to store both the materialized and the raw data. Table 3.4 shows the empirical values and theoretical estimates of μ for different settings. For both uniform and window-based sampling, the empirical and analytical computations yield similar values. Moreover, the empirical computation shows that the time-based sampling strategy performs better than the uniform sampling strategy. When the number of materialized feature chunks is 0 or 12000, the design

Table 3.4: Empirical computation and theoretical estimates (bold numbers) of μ for different sampling strategies and materialization rates ($\frac{m}{n}$). We omit the materialization rates 0.0 and 1.0 since both the empirical and theoretical estimates of μ are 0.0 and 1.0 for every sampling strategy.

	URL		**Taxi**	
Sampling	$\frac{m}{n}$=0.2	$\frac{m}{n}$=0.6	$\frac{m}{n}$=0.2	$\frac{m}{n}$=0.6
Uniform	0.52 (**0.52**)	0.91 (**0.91**)	0.51 (**0.52**)	0.90 (**0.91**)
Window-based	0.58 (**0.58**)	1.0 (**1.0**)	0.57 (**0.58**)	1.0 (**1.0**)
Time-based	0.68	0.97	0.65	0.97



(a) URL dataset.

(b) Taxi dataset.

Figure 3.8: Effect of the online statistics computation and dynamic materialization on the deployment cost.

of the deployment platform guarantees that μ is 0.0 and 1.0, respectively. Therefore, we do not report those results in the table.

To examine the effect of μ on the deployment cost, we plot the total deployment cost using different sampling strategies and materialization rates ($\frac{m}{n}$) for the URL and Taxi deployment scenarios in Figure 3.8. 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).

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 μ value than the other two sampling strategies (Table 3.4). 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.

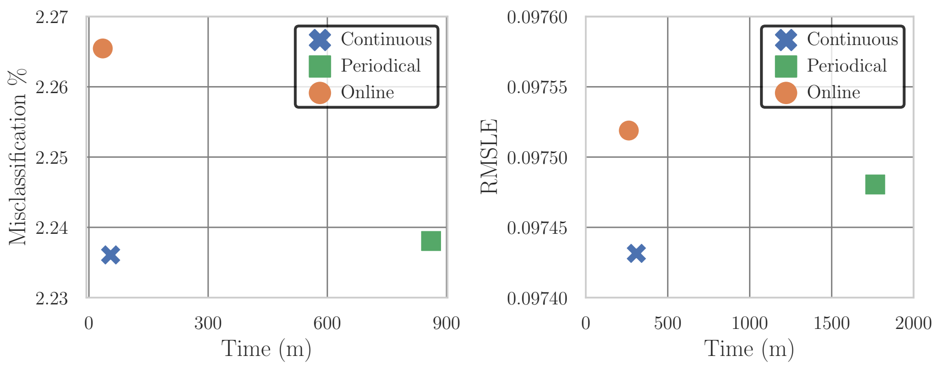
When the materialization rate is 0.6, window-based sampling has the best performance. Since the size of the window is smaller than the number of the materialized feature chunks, every sampled feature chunk is materialized. For the URL deployment scenario, window-based, time-based, and uniform sampling improve the performance by 40%, 36%, and 33%, respectively. For the Taxi deployment scenario, window-based, time-based, and uniform sampling improve the performance by 49%, 46%, and 37%, respectively. Similar phenomena explain the difference in the performance improvement at a materialization rate of 0.6 between the Taxi and the URL deployment scenarios. When the materialization rate is 0.6, more than 90% of the chunks are materialized. Therefore, the improvement in the Taxi deployment scenario is more noticeable due to 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 NoOpt in Figure 3.8). The extra disk access and data processing result in an increase of %110 for the URL (Figure 3.8a) and %170 for the Taxi deployment scenarios (Figure 3.8b) 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.

3.6.5 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 [51]. 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 3.9 shows the trade-off between the average quality and the total cost of the deployment. By utilizing continuous deployment, we reduce the total cost of the deployment by 6 to 15 times when compared with the periodical deployment, while providing



(a) URL dataset. **(b)** Taxi dataset.

Figure 3.9: Trade-off between average quality and deployment cost.

the same quality (even slightly outperforming the periodical deployment by 0.05% and 0.3% for the Taxi and URL datasets, respectively).

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 during the retraining process. However, in real-world scenarios, the training data and 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 [19]). 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.

3.7 Conclusion

In this chapter, we introduce the single-pipeline deployment setting and workloads. After a machine learning pipeline is designed and initially trained on a dataset, data scientists deploy the pipeline and make it available for answering prediction queries. To ensure that the model maintains 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, which provides the same level of model quality 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, the continuous deployment platform provides the same level of quality for the deployed model when compared with the periodical deployment approach.

The single-pipeline deployment setting applies to many real-world ML applications. However, in some cases, the deployment of one pipeline is not enough. In the next chapter, we introduce the multi-pipeline deployment setting and propose several optimization techniques to improve the execution of ML workloads in such a setting.

4 Multi-pipeline Deployment and Continuous Training

In this chapter, we introduce the multi-pipeline deployment setting and workloads. Several use cases, such as A/B testing and ensemble learning, require more than one pipeline to be deployed in parallel. In many applications, new training data continuously becomes available. A typical approach to ensure that the deployed ML models are up to date is to retrain the ML pipelines following a schedule, e.g., every day on the last seven days of data.

Existing solutions train and deploy one pipeline at a time, which generates redundant data processing since pipelines usually share similar operators. Our goal is to eliminate redundant data processing in production data science pipelines using materialization and reuse optimizations. In this chapter, we first categorize the generated artifacts of the pipeline operators into three groups, i.e., computed statistics, transformed data, and trained models. Then, we propose a solution to optimize the execution of the pipelines by materializing and reusing the generated artifacts. Our solution employs a materialization algorithm that given a storage budget, materializes the subset of the artifacts, which minimizes the run time of the subsequent executions. Furthermore, we offer a reuse algorithm that generates an optimal execution plan by combining the deployed pipelines into a directed acyclic graph (DAG) and reusing the materialized artifacts when appropriate. To evaluate our optimization techniques, we implement a system prototype. The prototype includes the following components:

1. Pipeline manager: a component for designing and deploying ML pipelines.
2. Materializer: a component for executing our materialization algorithm.
3. Data Store: a component for storing raw data and ML artifacts.
4. Optimizer: a component for executing our reuse algorithm and generating execution plans.
5. and Executor: a component for running the execution plan and training the ML pipelines.

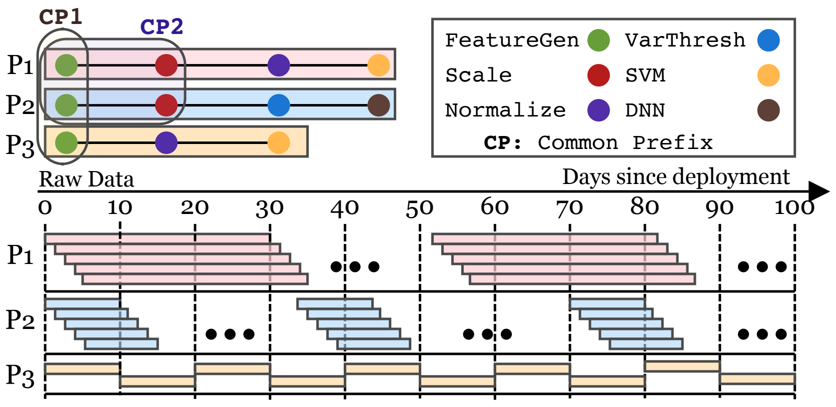


Figure 4.1: Execution of scheduled pipelines and the generated artifacts after every execution.

Our experiments show that our optimization techniques can reduce the training time by up to an order of magnitude for different deployment scenarios.

4.1 Motivation

Designing machine learning (ML) pipelines and training ML models are only the first steps in industrial ML settings. After the initial training, data scientists deploy the pipelines into a production environment to answer prediction queries. State-of-the-art solutions typically require *multiple* pipelines to maximize the prediction accuracy. Such practices are common in A/B testing [8, 9, 10], continuous integration of ML pipelines, where multiple pipelines are trained and tested before one is pushed to production [21, 22, 23], and automated or manual design and training of pipeline ensembles [11, 12].

In most enterprise applications, new training data arrives continuously. The current approach that ensures models stay up to date is to first utilize a lambda-style architecture [73] to store the continuously arriving data in a persistent storage unit. The stored data is partitioned by a time unit based on the timestamp of the data (e.g., hours or days). Then, the pipelines are retrained over past data intervals, e.g., a moving window over the last n days [18, 74]. Typically, the data scientists define the data interval and the frequency of the retraining. Retraining is paramount in online applications, such as ads click prediction [75] and mobile application recommendation [18], where the introduction of new products and users requires retraining of the pipelines to provide meaningful predictions.

Example Use Case. A mobile advertising company has a data collection system to store the served ads and their click outcome (clicked or not). The data arrives continuously, where each data point has a timestamp. The data collection system ingests the incoming data and stores the data partitioned by the time unit day of the timestamp. The data scientists build ML pipelines to train models that predict the likelihood of a click. They utilize ensemble learning and train three pipelines, i.e., p_1, p_2, and p_3, to combine their predictions. Figure 4.1 shows the executions of the pipelines in the first 100 days after the deployment. The pipelines consist of various components: feature generator (for generating polynomial and interaction features), standard scaler, normalizer, variance thresholding, and three ML models, i.e., SVM, DNN, and Logistic Regression (LR). CP represents a common prefix. For example, p_1 and p_2

have the common prefix `FeatureGen`$\rightarrow$`Scale` (CP2) and all three pipelines have the common prefix `FeatureGen` (CP1).

The data scientists retrain p_1 and p_2 daily on the last 30 and 10 days of data, and p_3 every 10 days on the last 10 days of data. Every pipeline execution results in three categories of artifacts: (1) the computed statistics, such as the mean and variance for the scaler, (2) the transformed features after every component, and (3) the trained models after every execution. For example, after the first execution, the feature generator and normalize components of p_1 generate polynomial and normalized feature artifacts in the interval $[0, 30)$. Similarly, the scale component of p_1 generates mean and variance statistics artifacts and scaled feature artifacts in the interval $[0, 30)$.

Problem. Although several existing systems provide end-to-end management (from training to deployment) of ML pipelines [18, 76, 19], they are only able to process one pipeline at a time. Therefore, in multi-pipeline scenarios, such as the one above, every pipeline is retrained in isolation. The pipelines in these scenarios are often very similar, and, thus, share many components. For example, data cleaning components [24], standardization and normalization of numerical data, and encoding of categorical data are standard techniques in the design of data science and ML pipelines. Furthermore, in many ensemble learning approaches, the data processing components of the pipelines are the same and only the ML models are different. In addition, current state-of-the-art solutions ignore the fact that it is the same group of pipelines that are retrained but in different intervals. Therefore, retraining pipelines in isolation results in redundant data processing, which leads to longer training times and, thus, delays in keeping ML models up to date.

Optimization Opportunities. By storing the generated artifacts (i.e., statistics, features, and models) in fast storage (e.g., RAM), we identify four possible opportunities to optimize the execution of multiple pipelines. (i) If we materialize the computed statistics, we can reuse them later to avoid redundant computation. For example, every execution of p_1 has a 29-day overlap with the previous execution. On the second execution (on the interval $[1, 31)$), since mean and variance can be computed incrementally, we reuse the mean and variance of the interval $[1, 30)$ and only compute the mean and variance of $[30, 31)$. (ii) Similar to statistics, if we materialize the intermediate features, we can reuse them in the subsequent executions. For example, the second execution of p_1 can reuse the generated features in the interval $[1, 30)$. (iii) By materializing the trained models, we can reuse them as initial points (warmstarting) for subsequent executions. (iv) Deployed pipelines may share similar components. Whether or not the input to these components is materialized, executing the pipelines one by one results in redundant component execution. Similar to multi-query optimization techniques [4], we can exploit common sub-expressions (pipeline prefixes in our case) to eliminate redundant data processing and loading of the materialized data. For example, if the result of the first two components of p_1 and p_2 are materialized, then, we can load them once and reuse them for both pipelines. Otherwise, p_1 and p_2 can share the execution of their first two components, when no artifacts are materialized. The first three opportunities exploit intra-pipeline, while the last one exploits inter-pipeline similarities.

4.2 Research Contributions

Inspired by materialized view selection [1] and multi-query optimization [4, 2, 3], we present a system for materializing and reusing generated artifacts to optimize the retraining of multiple deployed ML pipelines. We assume the input (training data) arrives continuously and is stored in a partitioned format based on a user-selected time unit. Each pipeline retrains on a moving window over the last n partitions. Our system employs a cost model that estimates the cost of the materialization and reuse of artifacts, materializes the artifacts that minimize the cost, and generates an execution plan to reduce the execution cost. In contrast to existing works on materialization and multi-query optimization, our system tackles the following challenges: (i) *Heterogeneity of the generated artifacts*: Generated artifacts can be features, statistics, and ML models. This is in contrast to traditional multi-query optimization where the output of each operator is only tuples. To enable materialization and reuse, we must determine an appropriate structure to represent multiple ML pipelines as well as unify their cost representation. (ii) *Limited storage*: Depending on the dataset size and complexity of the ML pipelines, it may not be possible to materialize all the generated artifacts due to storage constraints. To maximize the reuse opportunities for all the pipelines, we must associate the reduction in cost with the storage overhead. Our solution should also adapt to the changing workload characteristics, i.e., the pipelines and their schedules, as new pipelines are created, removed, or rescheduled. (iii) *Variety of execution plans*: Given the set of scheduled pipelines and existing materialized artifacts, there can be different ways of determining what (materialized) artifacts to load from storage, what artifacts to compute, and when to merge artifacts. We must devise a reuse algorithm that finds the optimal (according to our cost model) execution plan. (iv) *Data distribution change*: The distribution of incoming data may change. Our system must handle such distribution change to produce correct results and not reuse stale information.

In this chapter, we tackle the above challenges and make several contributions.

- We propose a system for optimizing the training of ML pipelines via materialization and reuse.
- We formulate the problem of materialization and reuse of the artifacts in ML pipelines and devise a unified cost model for all types of artifacts. We also propose an algorithm that computes the materialization benefit of artifacts under limited storage.
- We introduce a reuse algorithm that generates the optimal execution plan and handles distribution shift.
- Finally, we conduct an experimental evaluation with multiple datasets and ML pipelines to show that our system improves performance by up to an order of magnitude compared to current practices.

The rest of this chapter is organized as follows. In Section 4.3, we define pipeline DAG and artifacts of an ML pipeline. Section 4.4 describes the details of our implemented system. In Section 4.5, we formulate the problem of artifact materialization and propose our materialization algorithm. Section 4.6 presents our reuse algorithm. In Section 3.6, we evaluate the performance of our proposed optimization techniques. Finally, Section 4.8 concludes this chapter.

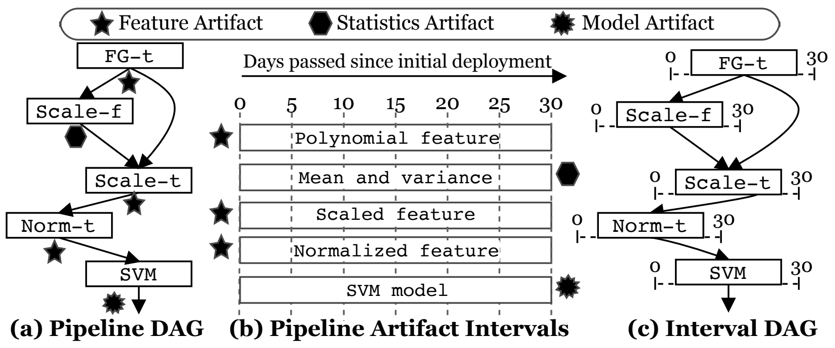


Figure 4.2: DAG, interval DAG, and artifacts of a pipeline.

4.3 Preliminaries

In this section, we define pipeline DAGs, the different artifact types generated after a pipeline execution, and how we reuse each type.

Pipeline DAG. Conventionally, a pipeline is a chain of operators. However, to provide support for the popular `fit/transform` API that is utilized in many data science libraries (e.g., scikit-learn [77] and SparkML [78]), we convert the pipelines represented in such systems into Directed Acyclic Graphs (DAGs). With such a DAG representation, our materialization and reuse algorithms can support not only unary operators, such as transformations, but also n-ary operators, such as aggregation, join, and union. The `fit` function, which computes some internal states (such as the mean and variance of a standard scaler), outputs the computed values. Figure 4.2a shows the DAG representation of pipeline p_1 of our example use case (Section 4.1). For the standard scaler, `Scale_f` and `Scale_t` show the `fit` and `transform` operators of the standard scaler.

Pipeline Artifacts. Executing a pipeline over an interval generates an *artifact*. We define three different types of artifacts: (1) Intermediate transformed data, which we refer to as the feature artifacts, are the result of data transformations such as `transform`, `join`, or `union`. (2) Computed statistics are artifacts that result from the `fit` or other aggregation operators. (3) ML models are artifacts that are resulting from training a model on features. Figure 4.2b shows the generated artifacts after the first execution of p_1 of our example use case on the interval $[0, 30)$ days. Feature and statistics artifacts are splittable. As a result, we can slice the artifacts into sub-intervals. To ensure that statistics artifacts are splittable, we only support incrementally computable statistics (or those statistics with an approximate incremental version, e.g., approximate quantiles [79]), as well as, algebraic and distributive aggregations [80]. However, model artifacts are not splittable. For example, in Figure 4.2b, there is only one SVM model in the interval $[0, 30)$, whereas, other artifacts can be split into sub-intervals (dashed lines going through an artifact means the artifact can be split into sub-intervals).

Interval DAG. In our solution, we utilize a modified version of the pipeline DAG, which we call *Interval DAG*. The vertices of the interval DAG encapsulate both the operators and the intervals that they operate on. Figure 4.2c shows the interval DAG of p_1 for its execution in the interval $[0, 30)$. We represent a vertex of the interval DAG by $v = \langle o, i \rangle$ where o is the operator and i is the interval (e.g., $\langle$`FG-t`$, [0, 30)\rangle$ in Figure 4.2c). The interval DAG has

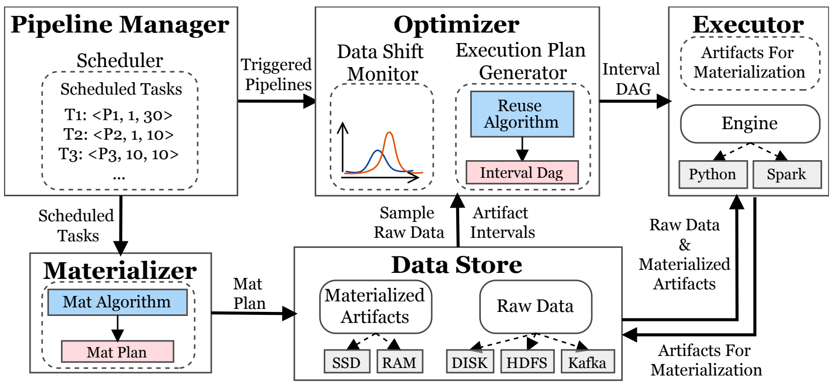


Figure 4.3: Workflow and architecture of the system.

different uses. For example, our materialization algorithm uses an interval DAG and assigns a benefit score to every vertex and the reuse algorithm generates an execution plan in the form of an interval DAG.

Reuse Procedures. We define two reuse procedures. Reusing feature and statistics artifacts involves loading the materialized sub-interval of artifacts from storage instead of computing them. As a result, reusing an artifact replaces its computation. For example, for the second execution of p_1 in interval $[1, 31)$, we can reuse the statistics artifacts (i.e., mean and variance) and the feature artifacts of the interval $[1, 30)$. Model reuse is the process of using a materialized model to *warmstart* the training procedure with the parameters of the materialized model. Therefore, reusing a model does not replace its computation, but reduces the training time [81, 82].

4.4 System Architecture

In this section, we first present the system architecture. Then, we discuss its extensibility for enabling the implementation of the user-defined pipelines, new runtime engines, and data stores.

4.4.1 System Components

Figure 4.3 shows the components and workflow of our system. Our system allows users to implement pipeline operators, design ML pipelines, and define schedules for them. Users can also add new scheduled pipelines, which we refer to as a *task*, and modify/delete existing pipelines or their schedules. In the rest of this section, we describe the different components and their functionalities.

Pipeline Manager. The pipeline manager keeps track of the submitted tasks. Each task specifies the ML pipeline, the slide (the execution frequency), and the interval (the size of the training data interval). The pipeline manager interacts with two other components, i.e., the materializer and the optimizer. Once users add, modify, or delete a task, the pipeline manager informs the materializer to (re)create a materialization plan. Furthermore, the pipeline manager checks the schedule of the tasks and—based on the slide values—triggers their execution accordingly. Whenever a set of tasks are triggered, the pipeline manager transfers the pipelines and the interval of the training data to the optimizer.

Materializer. Given the set of active tasks, the materializer component constructs a materialization plan using our novel materialization algorithm (Section 4.5). The plan specifies the artifacts with their intervals that the system should materialize and is passed to the data store. For example, the plan can indicate that the last ten days of mean and variance statistics, as well as, the normalized features (Figure 4.2) should be materialized after the execution of p_1.

Optimizer. The optimizer receives the triggered pipelines and their respective data intervals from the pipeline manager. The optimizer component has two main responsibilities, i.e., detecting distribution shifts and generating the execution plan.

When processing time-dependent data, it is vital to detect and handle distribution shifts. In such a scenario, reusing materialized artifacts may generate incorrect results. The data shift monitor is responsible for detecting a shift in the distribution. In our current implementation, we use the Kolmogorov-Smirnov test [83], but other techniques can be implemented as well. If a distribution shift in the data is detected, the data shift monitor notifies the execution plan generator to invalidate any affected materialized artifacts.

To generate the execution plan, the optimizer queries the data store for any materialized artifact. Given the materialized artifacts, the optimizer generates an execution plan that ensures the smallest execution cost based on our cost model. The optimizer utilizes our reuse algorithm (Section 4.6) to construct the execution plan. The execution plan is an interval DAG where every vertex specifies the interval of the data for processing and whether the data is materialized or not.

Executor. The executor component receives the execution plan (in the form of an interval DAG) from the optimizer and visits the vertices of the interval DAG in topological order. For a materialized vertex, it loads the artifact in the given interval from the data store. For an unmaterialized vertex, the executor runs the operator on the specified interval. After the execution, the executor sends the generated artifacts to the data store, which then decides what artifacts to materialize based on the materialization plan. The engine inside the executor is responsible for the actual execution. The engine is extensible and system users can implement new run times. In our prototype, we implement a Python and a Spark [84] execution engine.

Data Store. The data store is responsible for storing and managing the incoming raw training data and the materialized artifacts. It keeps the materialization plan constructed by the materializer and requests the executor to return the artifacts that are part of the plan. For detecting a distribution shift, the data store also provides a sample of the raw data in the requested interval to the optimizer. Furthermore, during the plan generation, the data store informs the optimizer of the existing materialized artifacts.

The data store can interface with different underlying storage systems, such as HDFS. In our system, new data continuously arrives and is partitioned based on a user-selected time unit (e.g., hourly or daily). The data store uses RAM for storing the materialized artifacts since we require fast access to them. However, the data store is agnostic to the underlying storage. Other fast storage types, such as SSD, can replace RAM for storing materialized artifacts.

Table 4.1: Table of notations.

Notation	Description
T, s, i	Scheduled task, slide, and interval
p	ML pipeline
o	Operator of a pipeline and its artifact
m_o, u_o	Materialized and unmaterialized splits of o
$C_{gen}(o, i)$	Cost of generating o in i
$C_{load}(o, m_o)$	Cost of loading the materialized split
$C_{exec}(o, u_o)$	Cost of executing the unmaterialized split
$C_{w\_exec}(o, i)$	Cost of training a warmstarted model
$C_{merge}(m_o, u_o)$	Cost of merging the splits

4.4.2 Extensibility

We design the system to be extensible. In general, there are two aspects of extensibility that we consider, i.e., new run times and new pipelines with different operators.

Engine and Data Store Extension. In our current implementation, we provide support for Python-based and Spark-based engines. Support for a new engine requires two steps. First, the user needs to implement the physical operators that are included in the physical plans in the new engine. Second, each engine may require the raw and the materialized artifacts to be in a specific format. Currently, we have a Numpy-based [85] data store for the Python engine and an RDD-based [86] data store for the Spark engine. Engines that require other types of data need a new data store as well.

Pipeline Implementation. Users can implement a new pipeline as a chain of operators, where every operator has a `transform` and/or `fit` method. The `fit/transform` APIs are common in data science libraries, allowing easy integration with existing tools. The only modification that users must make is to have the `fit` and model training operators return the computed statistics and trained models, which are then materialized in the data store.

4.5 Materialization

In real-world ML deployments, there can be many pipelines running on different data intervals. Since the input data size is typically large and the pipelines have many operators, the size of the generated artifacts becomes extremely large. In our experiments, we show that when the size of the raw data is 24 GB, the total size of the artifacts can reach 2 TB. Thus, storing every artifact is not cost-efficient. In this section, we first define our cost model and materialization problem. Then, we describe our materialization algorithm. Table 4.1 shows a summary of the notations used in this section.

4.5.1 Problem Formulation

Cost Model. We define a scheduled task as $T : \langle p, s, i \rangle$, where p is a pipeline DAG, s is the slide, which indicates how frequently we execute the pipeline, and i represents the interval of input data in the form $i = [begin, end)$. For example, the three tasks for our example use case in Section 4.1 are $T_1 : \langle p_1, 1, [0, 30)\rangle$, $T_2 : \langle p_2, 1, [0, 10)\rangle$, and $T_3 : \langle p_3, 10, [0, 10)\rangle$. When s is smaller than the size of i, consecutive executions of a pipeline operate on overlapping data intervals. This indicates that a sub-interval of data appears in several executions.

We denote a pipeline DAG p as the sequence of its operators (in topological order), $p = \langle o_1, o_2, ..., o_n\rangle$, where o_j $(1 \leq j \leq n)$ represents an operator. When executing a pipeline on the interval i, every operator o_j generates an intermediate artifact. We use o to also refer to the artifact generated by the operator. We denote the raw data by o_0 and the final ML model by o_n. An artifact can be divided into materialized and unmaterialized splits. For an arbitrary artifact o, we define m_o and u_o as the intervals where o is materialized and unmaterialized, respectively.

Because of the differences between the reuse procedures for model and non-model artifacts (described in Section 4.3), the cost of generating them is different. For simplicity, we use $C_{gen}(o, i)$ to represent the cost of generating both a model or a non-model artifact o in the interval i. Formula 4.1 presents the cost of generating a non-model artifact.

$$C_{gen}(o, i) = C_{load}(o, m_o) + C_{exec}(o, u_o) + C_{merge}(m_o, u_o) \tag{4.1}$$

$C_{load}(o, m_o)$ is the cost of loading the materialized split of o, if one exists. $C_{exec}(o, u_o)$ is the cost of executing o on the interval u_o, i.e., where o is unmaterialized. $C_{merge}(m_o, u_o)$ is the cost of merging the materialized and unmaterialized artifact splits of o. Note that for non-model artifacts, $m_o \cup u_o = i$ and $m_o \cap u_o = \emptyset$. Therefore, if a sub-interval of o is materialized, we do not need to recompute it.

Formula 4.2 shows the cost of generating a model artifact.

$$C_{gen}(o, i) = \begin{cases} C_{load}(o, m_o) + C_{w\_exec}(o, i) & \text{, if } m_o \text{ exists} \\ C_{exec}(o, i) & \text{, otherwise} \end{cases} \tag{4.2}$$

With or without reuse (i.e., model warmstarting), the model training process runs on the entire interval. When there is a materialized model artifact from an earlier execution, we load the model and execute the operator with a warmstarted materialized model. $C_{w\_exec}(o, i)$ represents the execution with a warmstarted model. If no materialized model exists, we execute the operator (train the model with random initialization of the parameters), which is denoted by $C_{exec}(o, i)$. Note that contrary to the non-model artifacts, the materialized interval m_o is not necessarily a sub-interval of i. This is because we can warmstart a model as long as the size and shape of the models (number of weight parameters) are equal.

The total cost of executing a pipeline p of n operators on interval i is the sum of the cost of generating every artifact of the pipeline.

$$C(p, i) = \sum_{o_j \in p} C_{gen}(o_j, i) \tag{4.3}$$

Problem Formulation. We define the problem of artifact materialization as finding the optimal set of artifact splits to materialize in order to minimize the sum of the execution cost of all the scheduled tasks under a limited storage budget. More formally, assume $\mathcal{T} = \{T_1, T_2, ..., T_{|\mathcal{T}|}\}$ is the set of scheduled tasks, where $T_t : \langle p_t, s_t, i_t\rangle$, $|p_t|$ is the number of artifacts in p_t, and β is the storage budget. We also define $\mathcal{M}$ as the set of all possible generated artifact splits of all the tasks. Then, the optimization problem is:

$$\begin{aligned} &\underset{\mathcal{M}^* \subseteq \mathcal{M}}{\text{minimize}} \sum_{t=1}^{|\mathcal{T}|} \frac{C(p_t, i_t)}{s_t} \\ &\text{subject to} \sum_{m \in \mathcal{M}^*} size(m) \leq \beta. \end{aligned} \tag{4.4}$$

The cost of a scheduled task is inversely proportional to its slide (the larger the slide the less frequently a pipeline is executed). Therefore, in Formula 4.4, we divide the execution cost of a pipeline by slide. For example, consider two tasks that have the same pipeline and interval, but different slides of 1 and 10, respectively. For every 10 executions of the first task, there is only one execution of the second task. Therefore, the overall cost of the second task is 10 times smaller. By dividing the cost by the slide, we can account for the impact of the slide in our optimization problem.

4.5.2 Benefit-based Materialization

The goal of our materialization procedure is to find the optimal set of artifact splits to materialize, i.e., the solution to the optimization problem in Formula 4.4. Given the large space of operators, cost, size, and scheduling intervals, the problem of finding the optimal set of artifact intervals is complex. A simpler scenario where only batch data are considered is shown to be NP-hard [42]. We propose a heuristic-based algorithm, where the main idea is to materialize artifacts with a high cost-to-size ratio. Thus, ensuring the result of costly operators that require a smaller storage space is materialized.

Our proposed materialization procedure comprises two steps. In the first step, for every scheduled task, we combine the interval DAG of each pipeline into a global interval DAG, which we refer to as the *Materialization DAG*. The materialization DAG is an interval DAG, where—besides the operator and interval—every vertex, v, contains the list of the pipeline identifiers they belong to, represented by $P(v)$. In the second step, we devise a greedy algorithm that at every step materializes a vertex that has the largest benefit score. We compute the benefit score in such a way that a higher score indicates a larger reduction in the execution cost if the artifact is materialized. Given the set of already materialized vertices, our greedy algorithm needs to recompute the scores. However, materializing a vertex does not affect the benefit score of every vertex in the materialization DAG. Therefore, to speed up the benefit recomputation, we propose an optimization that first detects the set of vertices that are impacted by the already materialized set and only recomputes the benefit of those vertices incrementally. The algorithm stops when the storage budget is exhausted.

In the rest of this section, we use the pipelines p_1 and p_2 from our running example of Section 4.1. We omit p_3 to simplify the description and the presentation of our materialization algorithm. For a scheduled task, the beginning and end of the interval are relative to the current system time, e.g., the interval $[0, 30)$ days shows the interval that started 30 days ago. The scheduled tasks of our running example are $T_1 : \langle p_1, 1, [0, 30) \rangle$ and $T_2 : \langle p_2, 1, [0, 10) \rangle$.

Materialization DAG Construction. The first step of the materialization procedure is to construct the materialization DAG. The procedure starts by sorting the tasks based on

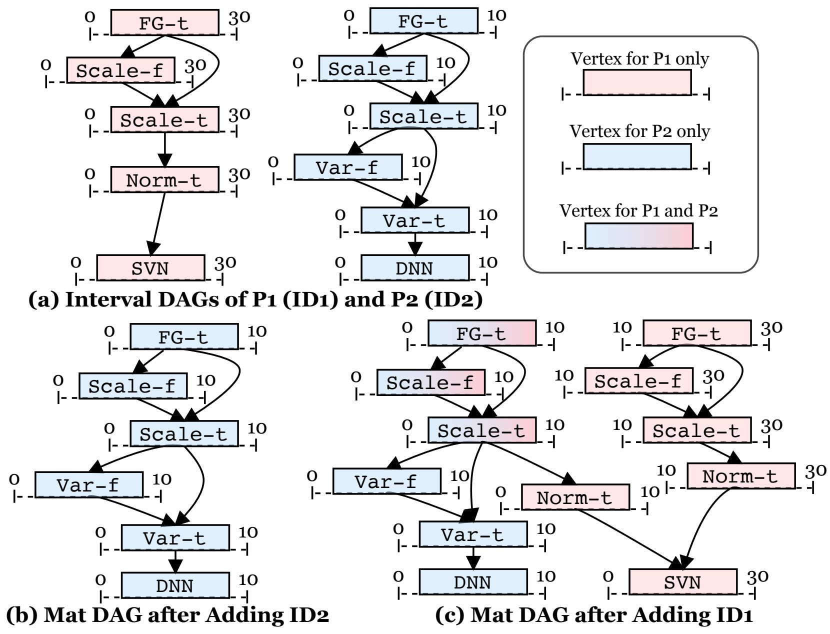


Figure 4.4: The process of materialization DAG construction.

their interval size (increasing order). Then, starting with an empty materialization DAG, the procedure adds sorted interval DAGs one by one. Figure 4.4a shows the interval DAGs of p_1 and p_2, which we refer to as ID_1 and ID_2. For every vertex of an interval DAG (starting from the source vertex), one of the following four scenarios might occur.

(S1) The vertex does not exist in the materialization DAG. In this scenario, we add the vertex to the materialization DAG and add edges to the parents of the vertex. Every vertex in Figure 4.4b belongs to S1. (S2) The vertex partially matches an existing vertex. I.e., the operator of the vertex exists in the materialization DAG, however, the interval is larger than the interval of the vertex inside the materialization DAG. In this scenario, we split the vertex into two. The first vertex has the exact interval as the existing vertex in the materialization DAG. Instead of creating a new vertex, we update $P(v)$ of the existing vertex in the materialization DAG by adding the pipeline identifier of the new vertex. Then, we add the second vertex to the materialization DAG. Figure 4.4c demonstrates an example of S2 (e.g., notice how $\langle \texttt{FG-t}, [0, 30) \rangle$ of ID_1 is split into $\langle \texttt{FG-t}, [0, 10) \rangle$ and $\langle \texttt{FG-t}, [10, 30) \rangle$ in the materialization DAG). (S3) The vertex does not exist in the materialization DAG, however, its parent was part of S2. In this scenario, we also split the vertex into two, in such a way that the resulting intervals match the intervals of the parent vertex. Figure 4.4c shows how $\langle \texttt{Norm-t}, [0, 30) \rangle$ of ID_1 is split into $\langle \texttt{Norm-t}, [0, 10) \rangle$ and $\langle \texttt{Norm-t}, [10, 30) \rangle$. (S4) The vertex is a last-level ML model. In this scenario, first, the vertex representing the model is added to the materialization DAG. Then, if the parent of the ML model was split earlier (because of S2 or S3), edges are drawn from all the vertices to the ML model.

Benefit Computation. In the second step of the materialization procedure, we utilize a greedy algorithm that iteratively computes a benefit score and materializes the vertex with the highest score. Let $v = \langle o, i \rangle$ be a vertex in the materialization DAG. We define the materialization benefit of v as the reduction in the cost of executing the pipelines when v is materialized. Formula 4.5 computes the materialization benefit of v.

$$\text{benefit}(v) = \frac{C(v)}{S(v)} * \sum_{p \in P(v)} L(p) \tag{4.5}$$

In Formula 4.5, $C(v)$ and $S(v)$ represent the estimated compute and storage costs of v. $C(v)$ is the sum of the execution cost of all the operators from source to v in the materialization DAG. $S(v)$ is the cost of storing the result of o in the interval i. The term $L(p)$ is the pipeline redundancy ratio. We define $L(p)$ as the number of times p processes a single data point, $L(p) = \frac{|i|}{s}$, i.e., size of the interval divided by the slide. For example, for pipeline p_1, the redundancy ratio is $\frac{30}{1} = 30$, since every day of the data will appear in 30 consecutive executions. Intuitively, an artifact interval with a high benefit value indicates one or both of the following cases. First, the cost-to-size ratio of the artifact is larger than that of the other artifacts; therefore, materializing it results in a larger relative improvement in execution cost. Second, the interval of the artifact has a high data processing redundancy; therefore, materializing it eliminates the redundancy and improves the execution cost. Once the algorithm computes the benefit of all the vertices in the materialization DAG, it materializes the vertex with the highest benefit score.

Benefit Recomputation. After materializing a vertex, the algorithm must recompute the benefit of the other vertices. We show that materializing a vertex *only* affects the benefits of its ancestors and descendants, and the algorithm only needs to adjust the benefits of such vertices.

Materialized vertices are directly loaded from the storage. Therefore, a materialized vertex may break the dependency of its descendants on the ancestors of the materialized vertex, which reduces the materialization benefit of the ancestors. Let v_m be the materialized vertex. If removing v_m from the materialization DAG disconnects the descendants of v_m from its ancestors, then we remove every pipeline p from $P(v_a)$, where $p \in P(v_m)$ and v_a is an ancestor of v_m. This is because the ML model in p where $p \in P(v_m)$ no longer requires the ancestors of v_m. This impacts the term $\sum_{p \in P(v)} L(p)$ in Formula 4.5.

For example, assume the algorithm has materialized the vertex $v_m = \langle \texttt{Var-t}, [0, 10) \rangle$ in Figure 4.4c. Since $P(v_m) = \{p_2\}$, the algorithm removes p_2 from all of its ancestors, i.e., $\langle \texttt{Var-f}, [0, 10) \rangle$, $\langle \texttt{Scale-t}, [0, 10) \rangle$, $\langle \texttt{Scale-f}, [0, 10) \rangle$, and $\langle \texttt{FG-t}, [0, 10) \rangle$. Note that materializing a vertex such as $\langle \texttt{Var-f}, [0, 10) \rangle$ does not break the dependency of its descendants (e.g., $\langle \texttt{Var-t}, [0, 10) \rangle$) from its ancestors (e.g., $\langle \texttt{Scale-t}, [0, 10) \rangle$), since there are more than one path connecting the ancestors to the descendants. Therefore, it has no impact on the benefit of its ancestors.

For descendants of a materialized vertex, we need to adjust the compute cost, i.e., $C(v)$. $C(v)$ is the sum of the execution cost of the vertices from source to v. If a materialized vertex breaks the dependency of its ancestors on its descendants, then the materialized vertex becomes a pseudo-source for its descendants. Thus, the algorithm recomputes the compute cost of all the descendants starting from the materialized vertex. For example, materializing $\langle \texttt{Scale-t}, [10, 30) \rangle$ in Figure 4.4c breaks the dependency of the vertices $\langle \texttt{Norm-t}, [10, 30) \rangle$ and $\langle \texttt{SVM}, [0, 30) \rangle$ on $\langle \texttt{FG-t}, [10, 30) \rangle$. Furthermore, if a materialized vertex does not break the dependency, it still impacts the total compute costs of its descendants. The execution cost of a materialized vertex is zero, which reduces the total compute cost of its descendants. For example, if $\langle \texttt{Var-f}, [0, 10) \rangle$ is materialized, we set its compute cost to zero before computing the cost of $\langle \texttt{Var-t}, [0, 10) \rangle$ and $\langle \texttt{DNN}, [0, 10) \rangle$. After the recomputation, the node with the highest benefit value is materialized. The greedy algorithm stops when the storage budget is exhausted.

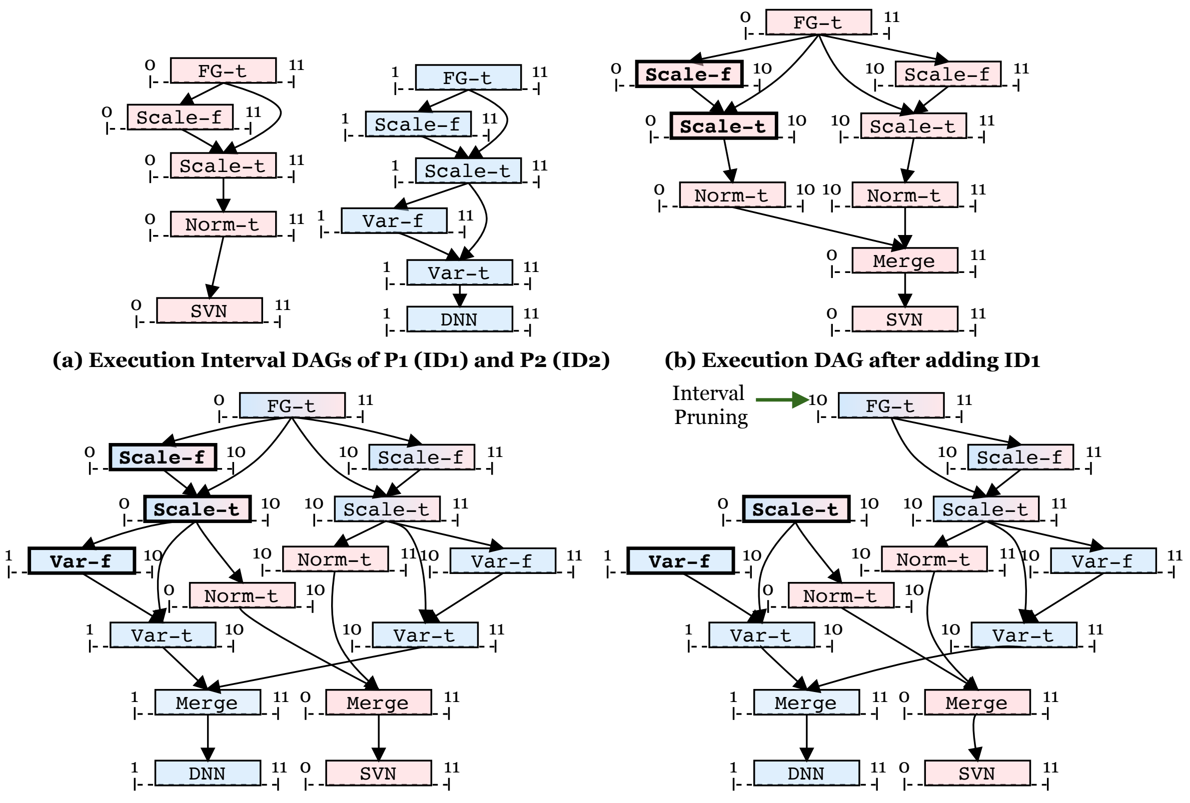
4.6 Reuse and Plan Generation

We now introduce our reuse and execution plan generation algorithm (for brevity, we refer to the algorithm as the *reuse algorithm*). We define the reuse problem as follows. Given a set of materialized artifacts (the output of the materialization algorithm of Section 4.5) and the set of triggered pipelines, i.e., the pipelines that are scheduled to execute next, find the optimal execution plan. The execution plan dictates what intervals of the materialized artifacts to reuse and what intervals the pipeline operators should be executed on.

The simplest approach to execute the scheduled pipelines is to run them one by one and reuse any materialized artifact from the artifact store. However, this results in redundant data processing and data loading, as many pipelines share similar operators. Our reuse algorithm follows three principles. First, it reuses materialized artifacts when possible. Second, inspired by multi-query optimization [3], our algorithm shares raw data, materialized artifacts, and the output of operators, whenever possible, to eliminate redundant data processing. Third, the algorithm ensures correctness in the presence of a distribution shift. In the rest of this section, we first describe the process of our reuse algorithm for generating an execution plan. Then, we describe the process of model reuse. Lastly, we explain how we handle distribution shifts in data.

4.6.1 Reuse Algorithm

The input to our reuse algorithm is a set of triggered pipelines and the output is an execution plan in DAG form, which we refer to as the *execution DAG*. The reuse algorithm consists of three main steps. First, the algorithm constructs the interval DAG for every triggered pipeline. The vertices of the interval DAG contain the operators and the actual interval of the data that will be used during the execution. All the vertices of the interval DAG of a triggered pipeline have the same interval. For example, if a pipeline will be executed on the interval $[1, 11)$, all of the vertices in the interval DAG will have the interval $[1, 11)$. Second, the algorithm sorts the interval DAGs by the size of their intervals in *decreasing order* and adds them one by one to the execution DAG. During this step, the algorithm consults the data store for available materialized artifacts and split the vertices into materialized and unmaterialized sub-intervals. Third, the algorithm traverses the execution DAG backward to prune the incoming edges of the materialized vertices. Materialized vertices are directly loaded from the artifact store, therefore, they are no longer computed. Furthermore, some sub-intervals of the vertices that have materialized descendants may not be necessary anymore. Therefore, the algorithm shrinks the intervals of such vertices. Figure 4.5 illustrates the main steps of our reuse algorithm, where the triggered pipelines are p_1 and p_2 of our running example. In the figure, we assume that eleven days have passed since the start of the system. Therefore, for p_2, the interval



(a) Execution Interval DAGs of P1 (ID1) and P2 (ID2)

(b) Execution DAG after adding ID1

(c) Execution DAG after adding ID2

(d) Execution DAG after vertex and interval pruning

Figure 4.5: The process of the reuse algorithm. Vertices with bold border and text indicate materialized vertices.

for training is $[1, 11)$ (the last 10 days) and for p_1, the interval is $[0, 11)$, since the scheduled interval of p_1 (30 days) is larger than the available data (Figure 4.5a). In this example, we assume that the artifact intervals `Scale-f` : $[0, 10)$, `Scale-t` : $[0, 10)$, and `Var-f` : $[0, 10)$ have been materialized.

Execution DAG Construction. The reuse algorithm starts with an empty execution DAG and adds the interval DAGs one by one. For every vertex of the interval DAG, if its sub-interval is materialized, then the algorithm splits the vertex into multiple vertices. Each new vertex represents a continuous sub-interval where the data is either materialized or unmaterialized. Figure 4.5b shows the execution DAG after adding ID_1. Since `Scale-f` : $[0, 10)$ is materialized, the vertex $\langle$`Scale-f`$, [0, 11)\rangle$ is split into $\langle$`Scale-f`$, [0, 10)\rangle$ and $\langle$`Scale-f`$, [10, 11)\rangle$, where $\langle$`Scale-f`$, [0, 10)\rangle$ is materialized (as shown by bold border in the figure). Similarly the vertex $\langle$`Scale-t`$, [0, 11)\rangle$ is split into two vertices. More formally, the algorithm splits a vertex $v = \langle o, i\rangle$ into a set of vertices, represented by $SPLITS(\langle o, i\rangle)$. Every vertex in $SPLITS(\langle o, i\rangle)$ is a tuple of three elements $\langle o, i', ms\rangle$, where o is the same operator as the original vertex, i' is a sub-interval of i, and ms is an indicator variable, which is one when the artifact in the sub-interval i' is materialized and zero otherwise. The resulting sub-intervals of the vertices in $SPLITS(\langle o, i\rangle)$ do not overlap and their union is equal to i.

It is important to note that the goal of the algorithm is to minimize the cost function in Formula 4.3 of Section 4.5.1. Splitting an interval into materialized and unmaterialized sub-intervals reduces the execution cost. However, according to our cost functions, loading the materialized interval and merging the intervals incur extra costs. Similar to our cost function, let C_{exec} be the cost of executing an operator in an interval, C_{load} be the cost of loading an artifact in a given interval, and C_{merge} be the cost of merging all the materialized and unmaterialized sub-intervals. The reuse algorithm only splits the intervals of a vertex when Inequality 4.6 holds. The inequality indicates that the total cost when using a materialized sub-interval, i.e., cost of loading the materialized sub-intervals, executing the operator in the unmaterialized sub-interval, and merging all the sub-intervals, should be smaller than the cost of executing the operator on the entire interval.

$$\begin{aligned} C_{exec}(o,i) > &\sum_{\langle o,i',ms\rangle \in SPLITS(\langle o,i\rangle)} \Big[C_{load}(o,i') * ms + C_{exec}(o,i') * (1-ms)\Big] \\ &+ C_{merge}(SPLITS(\langle o,i\rangle)) \end{aligned} \tag{4.6}$$

The algorithm continues adding every interval DAG. If the execution DAG is not empty, for every new vertex, one of the two scenarios occurs. (S1) No matching vertex exists in the execution DAG. In this scenario, the algorithm proceeds as earlier, i.e., adding the vertex to the execution interval DAG and splitting it if there are materialized sub-intervals and Inequality 4.6 holds. For example, since the interval `Var-f` : $[0, 10)$ is materialized, the vertex $\langle$`Var-f`$, [1, 11)\rangle$ of ID_2 is split into $\langle$`Var-f`$, [1, 10)\rangle$ and $\langle$`Var-f`$, [10, 11)\rangle$ in Figure 4.5c. (S2) There exists a matching vertex with full or partial interval overlap. In this scenario, the algorithm does not add new vertices since there are existing vertices that match the vertex. Since the algorithm adds vertices sorted by their interval in decreasing order, any interval of the current interval DAG is smaller or equal to the interval in the execution DAG. Therefore, the algorithm does not need to add or update the vertices. For example, the result of splitting $\langle$`Scale-f`$, [1, 11)\rangle$ of ID_2 already exist in the execution DAG ($\langle$`Scale-f`$, [0, 10)\rangle$ and $\langle$`Scale-f`$, [10, 11)\rangle$ in Figure 4.5c). Before training a model, the algorithm merges the sub-intervals of the last non-model operators.

Execution DAG Pruning. After adding all the interval DAGs of the triggered pipelines, the reuse algorithm traverses the execution DAG backward starting from the last non-model artifact and performs two types of pruning. First, the algorithm prunes edges and vertices from the execution DAG that are no longer necessary. Materialized artifacts are loaded from the data store and are independent of their predecessors. Thus, the reuse algorithm prunes the incoming edges of the materialized artifact. Vertices that have no incoming or outgoing edges are removed from the execution DAG. Such a scenario occurs when a materialized artifact, i.e., no incoming edges, only has materialized successors, i.e., no outgoing edges. The second pruning operation shrinks the sub-interval of the vertices that are not required. During the backward traversal, the reuse algorithm propagates the interval of every unmaterialized vertex to its predecessors. Then, for every vertex visit during the traversal, the reuse algorithm sets the interval of the vertex to the union of all the intervals propagated from its successors. Using this approach, only the interval of the data that is required by the downstream vertices will be computed.

Figures 4.5c and 4.5d show the execution DAG before and after pruning. Since $\langle$`Var-f`$, [1, 10)\rangle$ and $\langle$`Scale-t`$, [0, 10)\rangle$ are materialized, the algorithm prunes their incoming edges. Furthermore, vertex $\langle$`Scale-f`$, [0, 10)\rangle$ is completely removed from the DAG, since the vertex itself and its successor are materialized. Because of pruning some of the edges and vertices, only the interval $[10, 11)$ is propagated to vertex $\langle$`FG-t`$, [0, 11)\rangle$. Therefore, the algorithm prunes sub-interval $[0, 10)$ from the vertex resulting in the final pruned execution DAG.

Reusing Model Artifacts. We support model reuse through warmstarting the model training. When there are materialized models, the reuse algorithm initializes the training procedure with the weight parameters of the materialized model. If multiple models in the artifact store overlap with the model in the execution DAG, the algorithm selects the latest model for warmstarting.

4.6.2 Handling Distribution Shift

When a distribution shift occurs, reusing materialized artifacts may generate inconsistent results. This is particularly the case for operators that generate feature artifacts and receive statistics artifacts as input. Consider the case when scaled features resulting from standard scaling are materialized. The presence of a distribution shift indicates that the old materialized features were scaled with a different mean and variance. Thus, reusing such materialized feature artifacts generates inconsistent output that hurts the final accuracy of the ML models. Therefore, in such a scenario, the reuse algorithm only splits vertices until (and including) the *first* statistics-based operator (e.g., `fit` or other aggregation operators) of every path starting from the source vertex. This is because a distribution shift only impacts the statistics-based and model training components. Generated feature artifacts of the operators that are not statistics-based are agnostic to the change in the distribution. As a result, we can still reuse such materialized artifacts in future executions.

In Figure 4.5, the operator `FG-t` (polynomial feature generator) is not statistics-based, which makes it agnostic to a distribution shift. Similarly, the operator `scale-f`, which computes some statistics over generated features of `FG-t`, is not impacted by the distribution shift (since its input is agnostic to the distribution shift). However, a distribution shift affects the artifacts after the `scale-f` operator. Therefore, in our example, in the presence of a distribution shift, the reuse algorithm is only allowed to reuse the materialized vertex $\langle$`Scale-f`$, [0, 10)\rangle$.

4.7 Evaluation

In this section, we experimentally evaluate: (i) the end-to-end performance of our system when compared to baseline and state-of-the-art solutions, (ii) the effect on model accuracy even in the presence of a distribution shift, and (iii) the overhead of our materialization and reuse optimizations. Our results show that our solution achieves up to an order of magnitude better performance compared to current practices while maintaining the model accuracy even when data distribution changes. In addition, our optimizations add negligible overhead (less than 0.01%) to the ML pipelines.

4.7.1 Setup

Prototype and Environment. We implement a prototype of our system in Python 3.7 using two different execution engines: a single-node and a parallel execution engine. We

Table 4.2: Details of the datasets, number of pipelines, use case schedules, and dataset and artifact sizes. Ensemble and Interval Tuning are two use cases that we evaluate in the experiments. The respective columns show the task schedules of each use case. For the taxi dataset, we only evaluate the ensemble use case.

Dataset	# Pipelines	Ensemble	Interval Tuning	Data Size
Higgs [87]	6	Daily on the last last 7 days	Daily on the 3, 5, and 7 days	Raw = 4 GB Total Artifact = 100 GB
URL [20]	6	Daily on the last 7 days	Daily on the 7, 14, and 21 days	Raw = 2.2 GB Total Artifact = 4 GB
Taxi [88]	10	Daily on the last 6 months	- - -	Raw = 24 GB Total Artifact = 2 TB

implement the single-node execution engine in Python, where we extend several of the existing scikit-learn models and preprocessing components [77]. For our parallel execution engine, we utilize Spark 3.1.2 [84] and its ML library [78] to implement ML models and preprocessing components.

Our reuse algorithm relies on a `merge` method to combine the sub-intervals of feature and statistics sub-intervals. In the Python engine, we represent feature artifacts as Numpy arrays [85] and use the built-in `concatenate` method to merge them. In the Spark engine, we represent feature artifacts as RDDs [86] and use the built-in union function of Spark for merging feature artifacts. The implementation of the `merge` method for the statistics artifacts depends on the type of statistics. In our current implementation, we implement the `merge` method for several statistics, such as count, mean, and variance. For user-defined statistics-based operators, the user must provide the `merge` implementation.

We also utilize the built-in `persist` method of RDDs for materializing the artifacts in memory. Furthermore, to optimize storage requirements, we materialize the RDDs in serialized form, which is another built-in functionality of Spark.

For our single-node (Python) experiments, we use a Linux Ubuntu machine with 128 GB of RAM. For our Spark experiments, we utilize an 11-node (one master and 10 workers) cluster, with each node having an Intel Xeon 2.4 GHz 16 cores and 28 GB of dedicated RAM.

Systems and Baselines. We evaluate the performance of different deployment scenarios on our system and other baselines. *MaR* is our system, which includes our novel DAG-based materialization and reuse algorithms. *TFX* is an implementation of a deployment system based on TFX [18]. TFX executes the pipelines one by one and, for every pipeline, it materializes the results of the last step. *Baseline* represents the most common current practice where pipelines are executed one by one without any materialization or reuse.

Datasets and Pipelines. We use three datasets and design several ML pipelines for each one. For two of the datasets, we consider two common multi-pipeline use cases, i.e., an ensemble of pipelines and interval tuning. For the third dataset, we only show the results for the ensemble use case as the results for the interval tuning follow the same patterns. Table 4.2 shows the datasets, pipelines, and their schedules for each use case.

Higgs [87] is a static dataset for a classification task. We manually divide the dataset into 22 equal chunks, where each chunk represents one day and contains 500,000 data points. The total size of the raw data is around 4 GB. We design a total of 6 different pipelines to process the

Higgs dataset. Three pipelines include imputaion→scaling→SVM with varying regularization $\alpha = \{0.01, 0.001, 0.0001\}$. And the other three pipelines have imputation→polynomial feature generation→scaling→SVM with varying $\alpha = \{0.01, 0.001, 0.0001\}$. In the ensemble use case, we run all the pipelines every day on the last week (7 days) of the data. In the interval tuning use case, we select the two top-performing pipelines and execute them every day on the last 3, 5, and 7 days of data, respectively (resulting in a total of 6 different tasks). Depending on the ML pipelines, the total size of the intermediate artifacts can reach up to 100 GB. We contribute the large size of the intermediate artifacts mainly to the polynomial feature generation operator.

The *URL* dataset [20] is a collection of URLs—labeled as malicious or legitimate—and their lexical and host-based features gathered for 121 days. The dataset contains both numerical and categorical entries. We also design 6 pipelines for processing the URL dataset. The pipelines contain imputation→scaling→feature hashing→SVM model. There are three different hash sizes (10k, 25k, and 50k) and 2 different regularization values $\alpha = \{0.01, 0.001\}$. In our implementation, we utilize the feature hasher of scikit-learn to generate sparse features out of the categorical entries. The URL dataset contains a large number of features, i.e., between 10,000 to 50,000. Since we utilize sparse vectors to store the preprocessed features, the actual size of the generated artifacts is small (up to 4 GB). Similar to Higgs, in the interval tuning use case, we select the two top-performing pipelines and execute them every day on the last 7, 14, and 21 days. The URL dataset has distribution shifts, and we primarily use it to investigate the performance of our system under distribution changes. We use the single-node engine (Python) for running all the experiments for the Higgs and URL datasets.

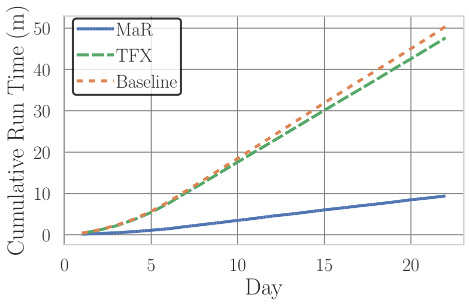
The *Taxi* dataset [88] contains records of Taxi rides in New York city. We use the portion of the data gathered in the year 2015 to evaluate our parallel execution engine (Spark). Inspired by a Kaggle competition [89], we utilize this dataset to design pipelines for estimating the trip duration of every taxi ride (a regression task). We design 10 pipelines based on some of the publicly available top-performing solutions of the Kaggle competition [90, 91, 92]. The pipelines have different operators such as date and distance extraction, anomaly detection, one-hot encoding, scaling, normalizing, and linear regression with different regularization values. The size of the raw data is roughly 24 GB and the total size of the intermediate artifacts after executing all the pipelines is approximately 2 TB.

4.7.2 End-to-end Performance

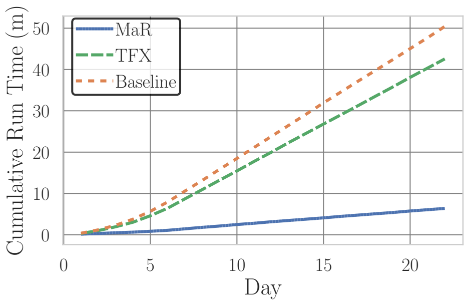
In this section, we analyze the end-to-end performance under varying budgets and number of parallel tasks. Here, we disable warmstarting and use the same random seed for the model training to ensure the training time remains the same, in order to focus on the impact of materializing and reusing statistics and feature artifacts. Therefore, we do not include the model training time in the reported figures.

Cumulative Cost With Different Budgets. Here, we investigate the end-to-end performance of running all pipelines together under varying budgets.

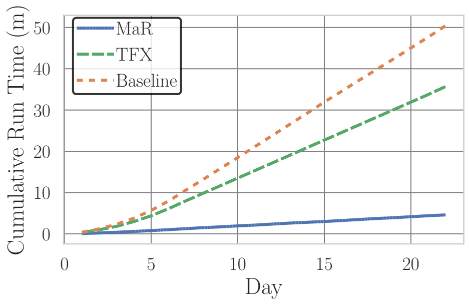
Figures 4.6 and 4.7 show the cumulative run time of both use cases (ensemble and interval tuning) for the Higgs dataset. Since the pipelines in the ensemble use case are highly similar (Figure 4.6), even with a small budget, MaR outperforms the baseline and TFX by a factor of 5.



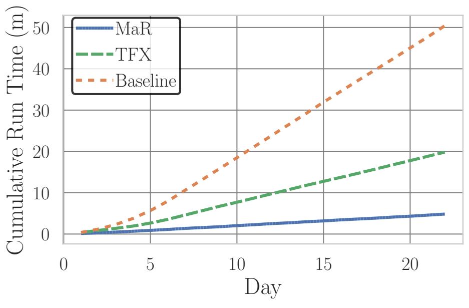
(a) Budget = 4 GB.



(b) Budget = 8 GB.

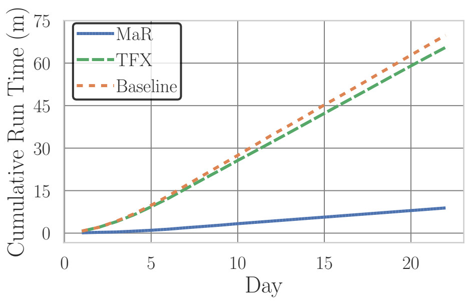


(c) Budget = 16 GB.

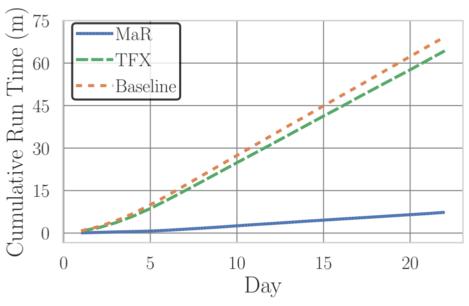


(d) Budget = 32 GB.

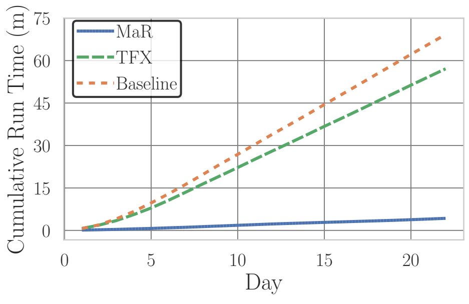
Figure 4.6: Cumulative run time for the Higgs ensemble use case.



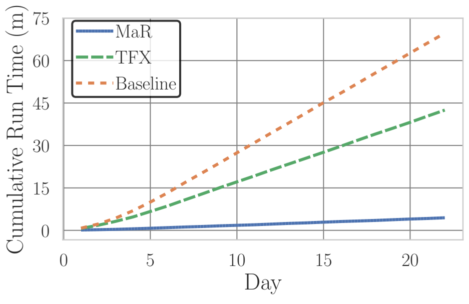
(a) Budget = 4 GB.



(b) Budget = 8 GB.



(c) Budget = 16 GB.

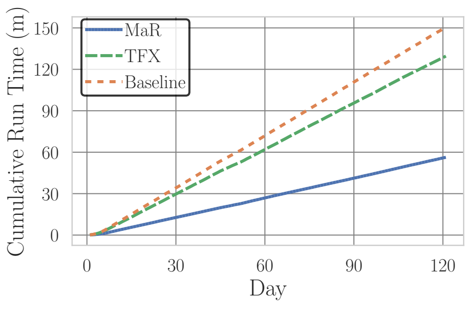


(d) Budget = 32 GB.

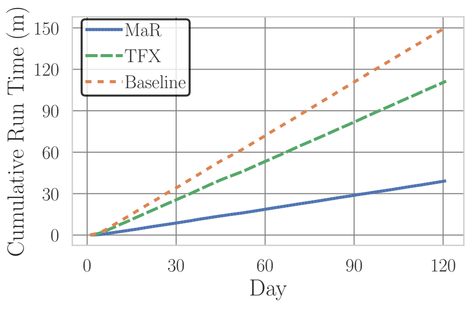
Figure 4.7: Cumulative run time for the Higgs interval tuning use case.

MaR identifies the two unique prefixes, i.e., imputation→scaling for the first three pipelines and imputation→polynomial feature generation→scaling for the second three pipelines (Table 4.2), and materializes their results only once. However, TFX materializes the generated artifacts of each pipeline separately, which quickly exhausts the budget. Therefore, with a small budget, TFX only materializes a small fraction of the artifacts and incurs a run time similar to the baseline. As the budget increases, TFX materializes a larger fraction of artifacts. When the budget is 32 GB, TFX materializes last-level features of all but one pipeline (it requires 38 GB to materialize the last-level features of all pipelines). Therefore, it outperforms the baseline. However, MaR still outperforms TFX by a factor of 4 thanks to its DAG-based execution, which shares operators' execution and loading of the materialized artifacts. The baseline does not depend on the materialization budget and incurs the same run time.

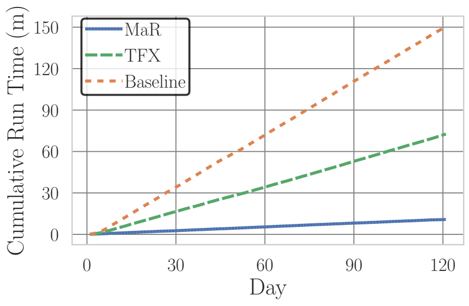
In the interval tuning use case (Figure 4.7), MaR outperforms both TFX and the baseline with a larger margin. With a budget of 4 GB, MaR outperforms the baseline and TFX by a factor of 8. For interval tuning, only the two top-performing pipelines with different intervals are used. The polynomial feature generator component, which is in the two top-performing pipelines, generates a large portion of the artifacts. Therefore, the total size of the generated artifacts for the interval tuning use case (approximately 100 GB) is larger than the ensemble use case (approximately 80 GB). Because of the larger size of the artifacts, TFX requires 52 GB to materialize the last-level features of the pipelines in the interval tuning use case (compared to 38 GB for ensemble use case). As a result, with a budget of 16 GB and 32 GB, the run time of MaR is 13 and 10 times smaller than TFX. MaR reaches its peak performance with a budget of 16 GB, i.e., MaR materializes all the artifacts that can be reused. However, with a budget of 16 GB, TFX only materializes a fraction of the artifacts.



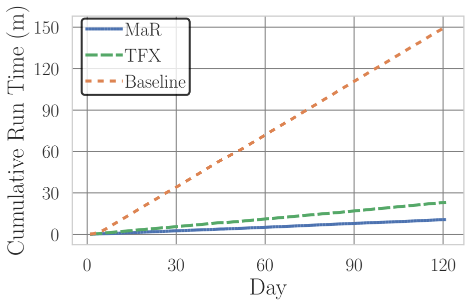
(a) Budget = 0.2 GB.



(b) Budget = 0.4 GB.



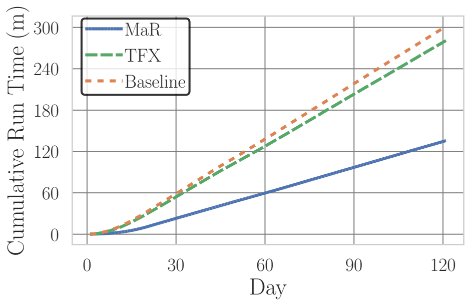
(c) Budget = 0.8 GB.



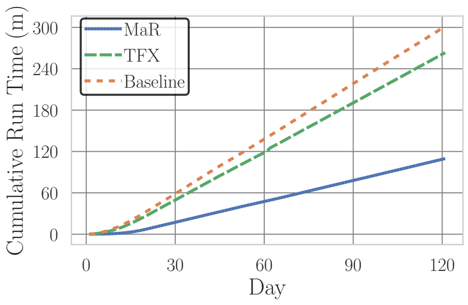
(d) Budget = 1.6 GB.

Figure 4.8: Cumulative run time for the URL ensemble use case.

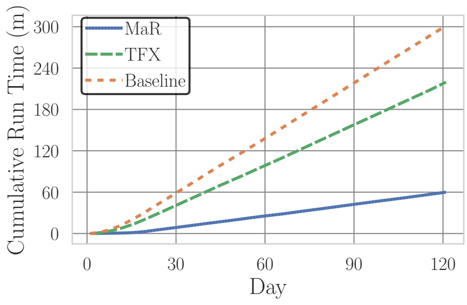
Figures 4.8 and 4.9 show the cumulative run times for the URL dataset for the ensemble and interval tuning use cases. We vary the budget from 0.2 to 1.6 GB, i.e., 5% to 40% of the total generated artifacts (similar to the Higgs dataset). In the ensemble use case (Figure 4.8), with a budget of 0.2 GB, MaR outperforms TFX and the baseline by factors of 2.5 and 3, respectively. For this use case, TFX can materialize all the last-level features with a budget of 1.6 GB. However, MaR still outperforms TFX by a factor of 2. Although TFX materializes everything, it still needs to compute the features for the newly available data for every pipeline separately. However, MaR utilizes a DAG-based execution where similar operators of different pipelines are executed once.



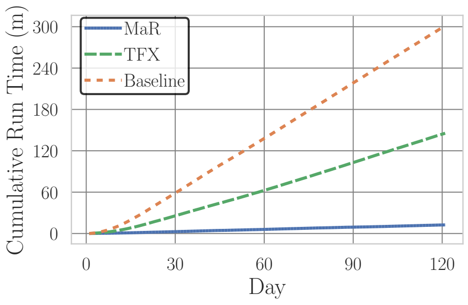
(a) Budget = 0.2 GB.



(b) Budget = 0.4 GB.

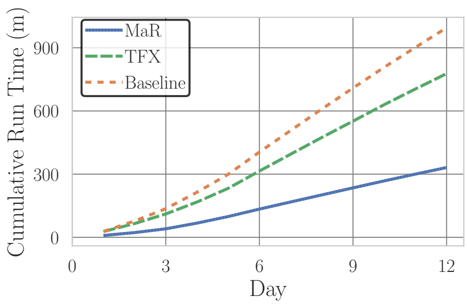


(c) Budget = 0.8 GB.

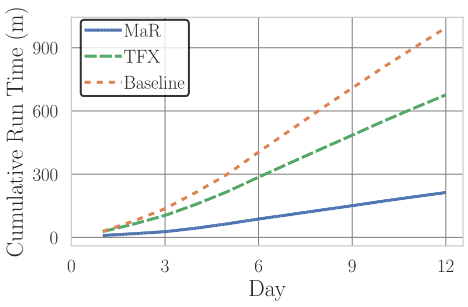


(d) Budget = 1.6 GB.

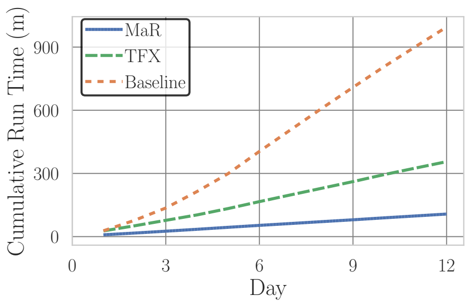
Figure 4.9: Cumulative run time for the URL interval tuning use case.



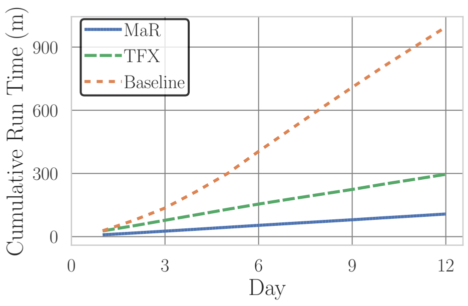
(a) Budget = 16 GB.



(b) Budget = 32 GB.



(c) Budget = 64 GB.

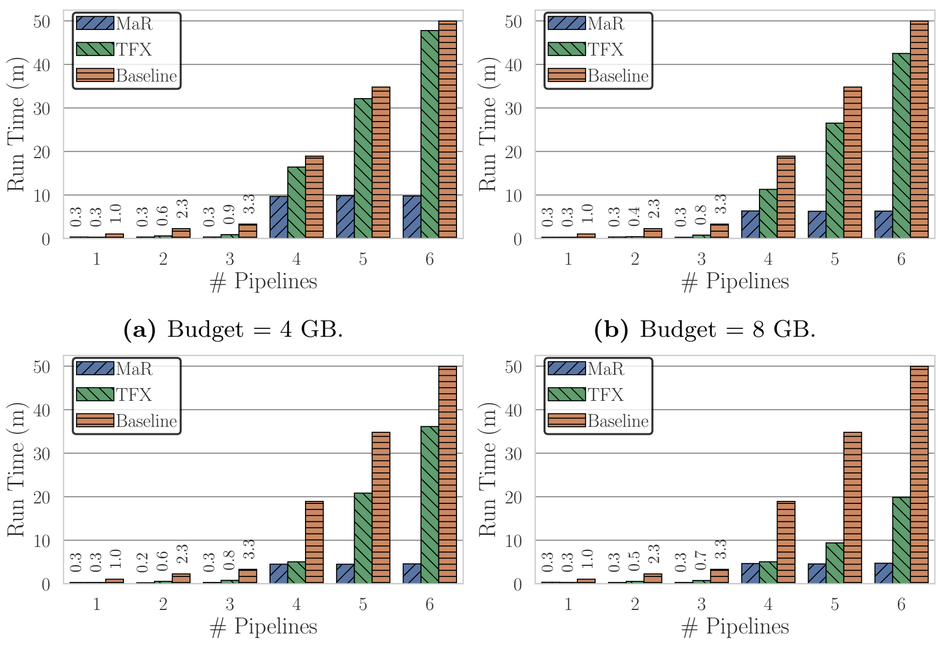


(d) Budget = 128 GB.

Figure 4.10: Cumulative run time for the Taxi ensemble use case.

Figure 4.9 shows that for the interval tuning use case, MaR outperforms TFX and the baseline by a larger margin when compared to the ensemble use case. There are two reasons for the better relative performance of MaR in the interval tuning use case. First, since intervals are larger (i.e., 7, 14, and 21 days), TFX requires 2.6 GB to materialize all the last-level feature artifacts (compared to 1.6 GB for the ensemble use case). Second, our materialization algorithm quickly materializes the sub-interval [0, 7), i.e., the last 7 days of the data. This sub-interval is shared between all the tasks, and materializing it greatly reduces the run time. TFX materializes the last-level feature artifacts of each task separately, which generates redundant data. When the budget is 1.6 GB, MaR nearly materializes all reusable artifacts. However, TFX only manages to materialize the artifacts of the first four pipelines and has to completely execute the two last pipelines on their entire intervals. As a result, MaR outperforms TFX by an order of magnitude with a budget of 1.6 GB.

Figure 4.10 shows the cumulative run times of the ensemble use case for the Taxi dataset. Similar to the other datasets, MaR outperforms the baseline and TFX by a factor of 2.5 and 3 when the budget is small (16 GB) and by a factor of 3 and 9 when the budget is very large (128 GB). Two reasons contribute to the better performance of MaR when compared to TFX. First, TFX requires a very large budget to materialize all the last-level features, i.e., 80 GB. Whereas, with a budget of approximately 40 GB, MaR can identify all the common pipeline prefixes and materialize them. Second, the DAG-based execution of MaR eliminates any redundant execution. As a result, even with a large budget of 128 GB—where TFX can materialize all the last-level features—MaR still outperforms TFX by a factor of 3.



(a) Budget = 4 GB.

(b) Budget = 8 GB.

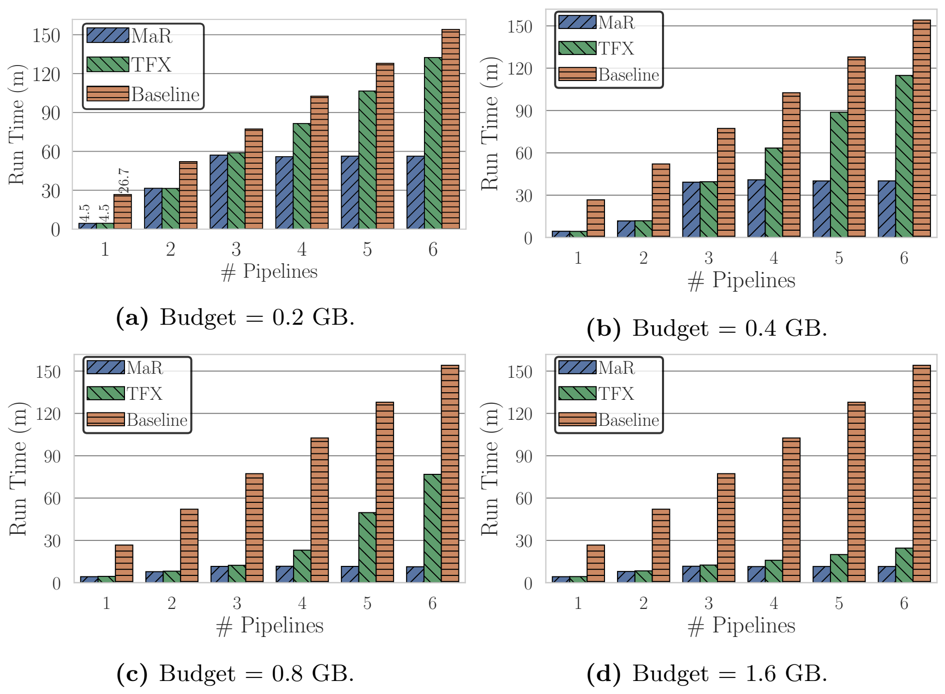
(c) Budget = 16 GB.

(d) Budget = 32 GB.

Figure 4.11: Total run time with different number of parallel tasks for the Higgs ensemble use case.

Number of Parallel Tasks. We now evaluate the performance of our system when increasing the number of parallel pipelines.

For the Higgs ensemble use case (Figure 4.11), the first three pipelines and the second three pipelines have similar data processing components (Table 4.2). Therefore, MaR has the same run time when the number of pipelines is 1 to 3 or 4 to 6, respectively. This is because MaR runs the DAG of the pipelines and shares the execution of similar operators. TFX, on the other hand, executes each pipeline separately. Therefore, with a budget of 4.0 GB, TFX executes all 6 tasks in around 48 min, 6 times larger than MaR's run time. When the budget



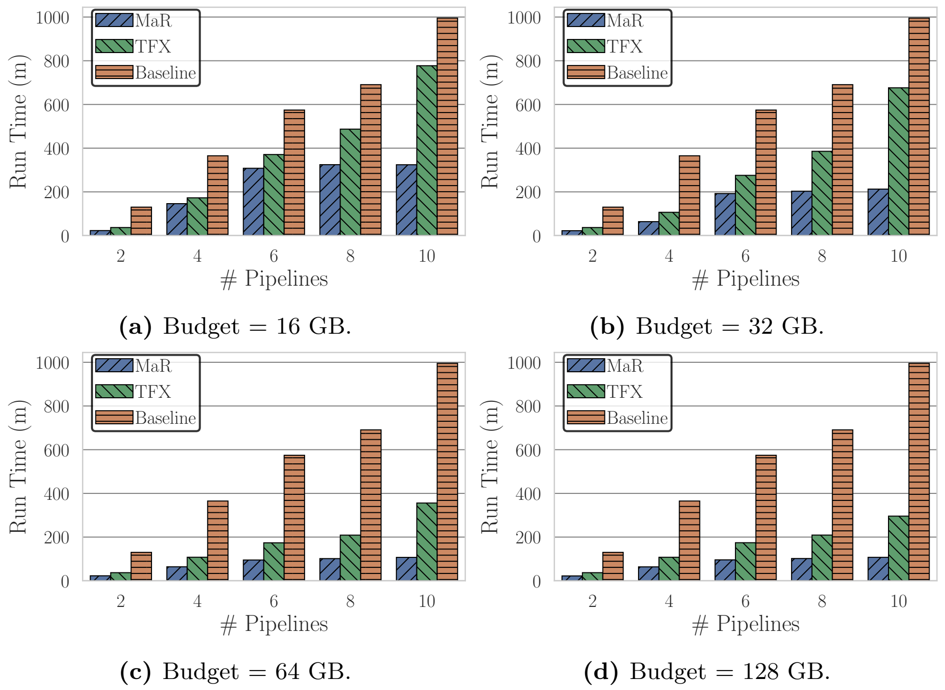
(a) Budget = 0.2 GB.

(b) Budget = 0.4 GB.

(c) Budget = 0.8 GB.

(d) Budget = 1.6 GB.

Figure 4.12: Total run time with different number of parallel tasks for the URL ensemble use case.



(a) Budget = 16 GB.

(b) Budget = 32 GB.

(c) Budget = 64 GB.

(d) Budget = 128 GB.

Figure 4.13: Total run time with different number of parallel tasks for the Taxi ensemble use case.

is 32 GB, TFX nearly materializes all of the last-level features. However, it still executes each task separately. Therefore, MaR outperforms TFX by a factor of 4.

For the URL ensemble use case (Figure 4.12), the data processing components of the first three pipelines are the same as the data processing components of the second three pipelines (Table 4.2). Therefore, MaR has the same run time when executing 3, 4, 5, or 6 pipelines, regardless of the materialization budget. However, as TFX executes the pipelines separately, even with a budget of 1.6 GB where all the last-level features are materialized, it incurs a total run time twice higher than MaR. MaR has the best relative improvement compared to TFX when the budget is 0.8 GB. This is because, at this budget, MaR can materialize all the reusable artifacts while TFX materializes the generated features of the first 4 pipelines. As a result, when the budget is 0.8 GB and the number of parallel pipelines is 6, MaR outperforms TFX by a factor of 6.

We observe a similar pattern for the Taxi ensemble use case (Figure 4.13). Thanks to the DAG-based execution, MaR outperforms TFX and baseline by a factor of 2.5 and 3 for small budgets (e.g., 16 GB), and by a factor 3 and 9 for large budgets (e.g., 128 GB).

For the interval tuning use case for the URL and Higgs datasets, we observe a similar trend. Since the size of the artifacts in the interval tuning use case is larger, the relative performance improvement of MaR is even greater than in the ensemble use case. For example, with the maximum budget and 6 parallel pipelines, MaR outperforms TFX by one order of magnitude for both URL and Higgs interval tuning use cases.

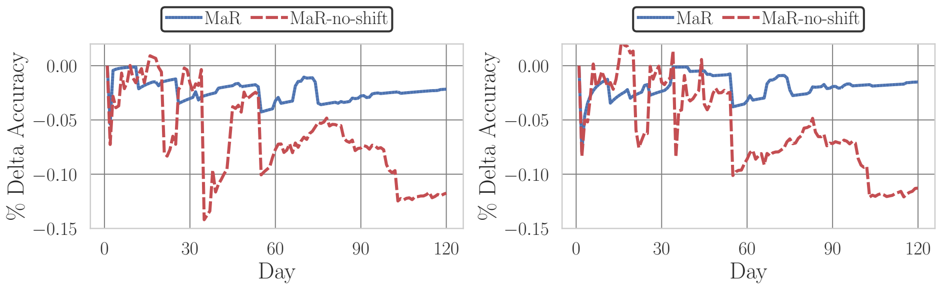
4.7.3 Impact of Distribution Shift

In this experiment, we enable distribution shift detection for MaR and examine the run time and quality of the models for URL data.

Accuracy. To show the impact of the distribution shift, in Figures 4.14a and 4.14b, we plot the difference between the accuracy of MaR, with and without shift detection, to the baseline approach. We assume that since the baseline does not reuse any materialized data, the distribution shift does not impact its performance. For the URL ensemble use case, we show the performance of the best pipelines (i.e., p_2 and p_3). We use the prequential technique [93] to report the accuracy of the models, i.e., we evaluate the model on the next day of data. The URL dataset is unbalanced, therefore, we show the difference in the running *balanced accuracy*[1]. With shift detection, the accuracy of the models is close to the baseline (average of 0.02% difference). Note that the model performance depends on the underlying shift detection technique. A technique that guarantees to find a distribution shift improves the accuracy. MaR with no shift detection reduces the model accuracy by 0.15%. This is because as the distribution of the data changes, reusing the materialized features generates inconsistent results, which negatively impacts the model performance.

Run time. Assuming that one implements the same shift detection mechanism in TFX so that the accuracy does not drop, we compare the run time of our system with TFX. Note that the original TFX system includes a mechanism for detecting distribution shifts. However, it requires the user to manually invalidate the materialized artifacts. Figure 4.15 shows the run time (data processing components) of MaR and TFX, with shift detection, for the URL ensemble use case. Out of 121 days, 70 days contain distribution shifts. As a result, for 70 days, no reuse was performed. MaR outperforms TFX and baseline by a factor of 2.3 and 3, respectively. We mainly contribute this improvement to the DAG-based execution of the pipelines since in nearly more than half of the days no reuse was performed.

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html`



(a) Pipeline p_2.

(b) Pipeline p_3.

Figure 4.14: Distribution shift effect on the accuracy of the URL ensemble use case.

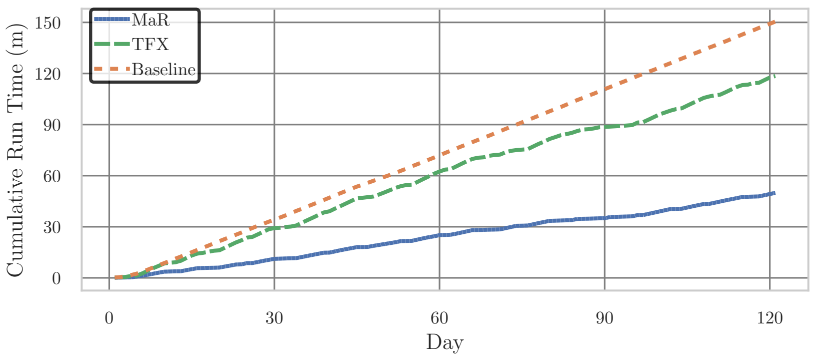
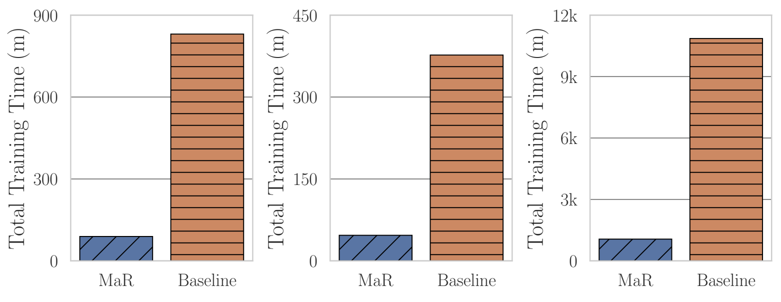


Figure 4.15: Impact of the distribution shift on the run time of the URL hyperparameter tuning use case.

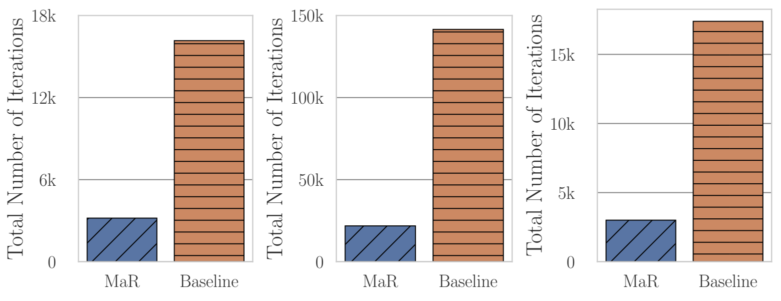
4.7.4 Impact of Model Reuse

We facilitate model reuse via warmstarting the model training. The main advantages of MaR for warmstarting are: (1) MaR offers warmstarting out-of-the-box, and in multi-pipeline scenarios, it automatically selects the matching model for warmstarting. (2) In the presence of a distribution shift, MaR invalidates the materialized models and chooses not to warmstart the models. TFX also offers warmstarting. However, it requires the user to select a model from the set of stored models. Therefore, the impact of warmstarting for both TFX and MaR is similar and we only show the results for MaR.

In Figure 4.16, we show the impact of warmstarting on the total training time (we exclude the data processing time) and the total number of training iterations for Higgs, URL, and Taxi ensemble use cases. The values show the total sum of training time and iteration counts for all pipelines for the entire deployment period. Since model artifacts are typically smaller than the feature artifacts and require large compute costs, our materialization algorithm assigns a large benefit score to the models and always materializes them. In the figure, we only show the total training time for budgets of 4 GB, 0.2 GB, and 16 GB for Higgs, URL, and Taxi, respectively. This is because larger materialization budgets have no impact on the training time. Model warmstarting using a previously converged model greatly reduces the number of iterations until convergence, as shown in Figures 4.16d, 4.16e, and 4.16f. As a result, MaR outperforms the baseline by up to one order of magnitude for all three use cases. Note that the impact of warmstarting is sensitive to the choice of hyperparameters such as the learning rate, convergence tolerance, and the maximum number of iterations. For example, in our Higgs ensemble use case experiments, we set the convergence tolerance to 10^{-5}, maximum



(a) Higgs run time. **(b)** URL run time. **(c)** Taxi run time.



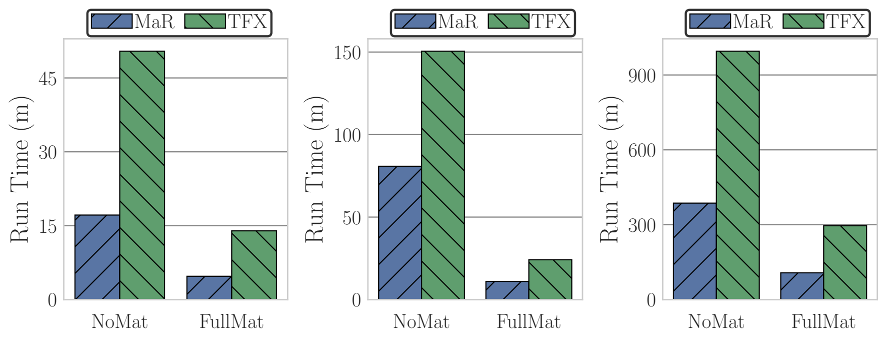
(d) Higgs iterations. **(e)** URL iterations. **(f)** Taxi iterations.

Figure 4.16: Effect of warmstarting on the training time and total number of iterations until convergence for the Higgs, URL, and Taxi ensemble use cases.

number of iterations to 1000, initial learning rate to 0.01, and set the learning rate to decrease as the number of iterations increase (i.e., the *invscaling* option of the SGD classifier of the scikit-learn package). A different set of configurations, such as a larger convergence tolerance or a smaller number of iterations, may result in a smaller gap between the training time of MaR and baseline.

4.7.5 Optimizations Impact

Impact of the Reuse and Plan Generation Algorithm. In this experiment, we analyze the impact of our reuse algorithm on the run time. We run the ensemble use cases in two scenarios. In the first scenario, we compare our system and TFX without any materialization (shown as NoMat in Figure 4.17). Figure 4.17 show the impact of our DAG-based reuse algorithm. Sharing the operators of the pipelines alone can increase the performance by a factor of 2 to 3 when compared to TFX. This is because, in the ensemble use cases, the pipelines share similar components. TFX without materialization has to execute each pipeline in isolation with no computation sharing potential. However, MaR combines the pipelines and executes the similar operators only once. In the second scenario, we compare our system with TFX when the materialization budget is unlimited, i.e., all the artifacts are materialized (shown as FullMat in Figure 4.17). In such a scenario, we show that MaR outperforms TFX by a factor of 2 to 3, even when the materialization budget is unlimited. In our experiments, we use a limited number of pipelines (six pipelines for Higgs and URL, ten pipelines for Taxi). The addition of new pipelines increases the run time of TFX linearly. However, depending on how similar the pipelines are, MaR can outperform TFX by a larger margin (we also confirmed this in Figures 4.11, 4.12, and 4.13).



(a) Higgs. (b) URL. (c) Taxi.

Figure 4.17: Total run time with no materialization and full materialization for the Higgs, URL, and Taxi hyperparameter tuning use cases.

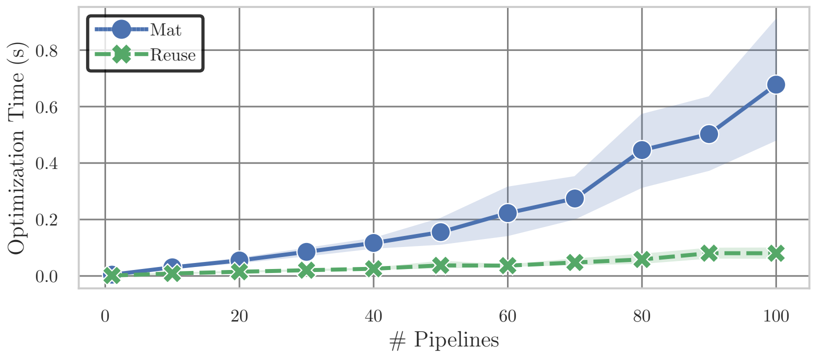


Figure 4.18: Overhead of the materialization and reuse algorithms with increasing number of parallel pipelines.

Optimization Overhead. In our use cases, we use a maximum of 10 pipelines. In these cases, the overhead of the materialization and reuse algorithm was less than 10 milliseconds. However, one can contribute the low overhead to the small number of parallel pipelines. To investigate the overhead of our optimizations, we simulate several workloads of up to 100 pipelines. For every pipeline, we randomly select the number of components from the interval $[4, 12]$ (We found this to be a common number based on the pipelines we observe in the Kaggle competition). Figure 4.18 shows the overhead of the materialization and reuse algorithm as a function of the number of parallel pipelines. Since we randomly generate the pipelines, we repeat the experiment 10 times and report the average (with error). Even with 100 parallel pipelines, our materialization and reuse algorithms incur an overhead of 600 and 100 milliseconds, respectively. In our experiments, pipelines have run times between 500 to 5000 seconds. Therefore, even with a large number of pipelines, our optimizations generate a negligible overhead of less than 0.01%.

4.8 Conclusion

In this chapter, we introduce the multi-pipeline deployment setting and workloads. We present a system prototype for optimizing the execution of multiple ML pipelines on overlapping data intervals. Our solution comprises a materialization algorithm that assigns a benefit score to the generated artifacts of the pipeline operators. The materialization algorithm stores the artifacts with the largest benefit in the artifact store. The benefit score directly correlates to the reduction in the execution cost. Furthermore, we propose a DAG-based reuse and execution plan generation algorithm that considers both the set of materialized artifacts and the set of all the scheduled pipelines. The reuse algorithm generates a plan that minimizes the execution cost by both loading the materialized artifacts and sharing the results of similar operators of different pipelines. We experimentally show that our materialization and reuse algorithms can reduce the execution cost of ML pipelines up to one order of magnitude and seamlessly adapt to the changes in the data distribution.

5 Related Work

In this chapter, we present the related work. We divide the related work into two categories. The first category includes existing solutions and systems that have the goal of simplifying the ML application design. Furthermore, such solutions provide support for the different stages of the ML application life cycle. The second category comprises solutions that utilize materialization and reuse optimization techniques, either in database systems or ML systems, to improve the performance of workload execution.

5.1 Systems for ML and Data Processing Application Life cycle

In this section, we describe existing works that provide support for the ML and data processing applications life cycle. Such works lead us to identify the three settings in ML applications life cycle (i.e., collaborative, single-pipeline deployment, and multi-pipeline deployment settings).

5.1.1 Collaborative Data Science Platforms

Cloud-based systems such as AzureML [28], Google's AI platform [94], Kaggle [14], and Google Colaboratory [15] provide the necessary tools for users to write ML workloads in Jupyter notebooks. Furthermore, users can publish and share their notebooks with others, which could result in higher quality workloads.

OpenML [29], ModelDB [30], and MLflow [95] are platforms that store ML artifacts, such as models and intermediate datasets, in a database [96, 31]. These platforms provide APIs for users to query the details of the ML artifacts.

Based on these works, we devise the collaborative setting. However, existing works only manage the storage of the generated artifacts with rudimentary support for artifact search. For example, OpenML stores ML pipeline programs and execution logs. Users can search through the programs and execution logs to analyze the performance of different pipelines. On the contrary, our solution utilizes the stored ML artifacts (i.e., ML models, generated features, and computed statistics) to optimize the execution of ML workloads via materialization and reuse.

5.1.2 Data Provenance Management

DataHub [97, 42], Context [98], Ground [99], ProvDB [100], Aurum [101], and JuNEAU [102] are data management and provenance systems that efficiently store fine-grained lineage information about the data artifacts and operations. We design our Experiment Graph, which stores the generated artifacts in collaborative settings, by utilizing the approaches discussed in these systems. Specifically, we follow DataHub's graph representation. However, contrary to these systems, we utilize the stored information to optimize the execution of the ML workloads.

Our materialization algorithms extend the materialization approach of Bhattacherjee et al. [42]. In collaborative settings, we extend the existing materialization algorithm to tailor it to ML workloads by considering several ML-specific aspects of the generated artifacts. Furthermore, we devise a deduplication strategy that drastically reduces the size of the materialized artifacts. In multi-pipeline deployment settings, we extend the existing materialization algorithm to support streaming data and time intervals (instead of batch datasets).

5.1.3 ML Model and Pipeline Deployment

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 [103], Velox [19], Clipper [104] , and TensorFlow Extended [18] have proposed architectures that also consider model deployment and prediction query answering. Such works lead us to identify the single-pipeline and multi-pipeline deployment settings and workloads.

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 solution works in a dynamic environment where training data, prediction queries, and ML workloads continuously arrive.

Velox is an implementation of the common periodical retraining 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 retrains the model using Apache Spark [84]. However, Velox has several 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. Forth, the platform only deploys the final model and does not support the deployment of the machine learning pipeline. Lastly, Velox only supports the deployment of one pipeline at a time, which makes it inapplicable to multi-pipeline deployment settings. In single-pipeline deployment settings, our approach differs from Velox as it exploits the underlying properties of SGD to integrate the training process into the platform's workflow. Our solution replaces offline retraining with proactive training. As a result, our solution 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, Clipper does not update the deployed models, which over time leads to outdated models. On the other hand, our solution focuses on the maintenance and continuous update of the deployed models and pipelines.

TensorFlow Extended (TFX) is a platform that supports the deployment of ML 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 warmstarting 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. In the single-pipeline deployment setting, our solution performs more rapid updates to the deployed model. By exploiting the properties of the SGD optimization technique, our deployment platform rapidly updates the deployed models (seconds to minutes instead of several days or weeks) without increasing the overhead. Furthermore, TFX only supports the deployment of one pipeline at a time, which renders it inapplicable to multi-pipeline deployment settings.

5.2 Materialization and Reuse Optimizations

Materialization and reuse optimizations techniques are commonly utilized in many database and ML systems to optimize the execution of workloads. In this section, we provide an overview of the existing efforts in both database and ML systems. Furthermore, we discuss how the existing solutions differ from our solutions.

5.2.1 Materialization and Reuse in Database Systems

Multi-query optimization (MQO) and view materialization in database systems, which have the goal of optimizing the execution of multiple SQL queries, either by sharing the execution of common sub-expressions or finding a common set of views to materialize, have been studied for over 30 years [4, 3, 1, 2]. More recent works, such as Nectar [105], ReStore [106], MRShare [107], and the work of Wang et al. [108], utilize MQO and materialization and reuse of intermediate data in parallel execution frameworks such as MapReduce [109] and DryadLINQ [110].

Existing works can be divided into three groups. The first group [105, 106] solely rely on materializing and reusing sub-expressions, e.g., the result of map or reduce functions, with no MQO. The second group only offers MQO by sharing the scan or map outputs [107]. And the third group offers MQO and materialization by both sharing scans and caching the result of map and reduce operators [108]. Some works on streaming data also utilize materialization and reuse to optimize the execution of queries [111, 112, 113, 114]. However, existing streaming-based solutions only share/reuse intermediate data for aggregation and join queries. In the single-pipeline and multi-pipeline deployment settings, our solution differs from such approaches, as we focus on MQO and materialization and reuse to optimize the execution and retraining of ML pipelines on *overlapping intervals of time-ordered data*. In addition, our approach takes *heterogeneous intermediate results* into account (i.e., ML models, statistics, and features), in contrast to the previous approaches that deal only with tuples and cannot be trivially used in our setting.

5.2.2 Materialization and Reuse in ML Systems

Several ML systems utilize materialization and reuse optimization techniques to improve the execution performance of ML workloads. Particularly, they offer materialization and reuse in workloads such as iterative ML [6], model diagnosis [39], end-to-end ML [115], approximate ad-hoc exploratory ML [116, 117], interactive feature selection [5], and knowledge base construction [118]. These systems have several fundamental differences to our solutions. First, the workload DAGs are typically small as these systems work with small ML pipelines. Therefore, these systems do not need to tackle the problem of searching for reuse opportunities in a large graph. This renders them in-applicable to collaborative settings where Experiment Graph is large. Helix is the only system that offers a polynomial-time reuse algorithm, which has a higher overhead when compared to our linear-time reuse algorithm in collaborative settings. Second, the materialization decisions in these systems only utilize run time and size and do not take into account the model quality. Third, our solution operates in collaborative and multi-pipeline settings. Whereas, the scope of optimization in these systems, except for Mistique [39], is limited to a single session or pipeline. However, Mistique is a model diagnosis tool, which enables users to query intermediate artifacts from an artifact store efficiently. Whereas, we focus on automatically generating optimal execution plans for future workloads by reusing stored artifacts. Furthermore, such works do not take into consideration the problem of optimizing the execution and retraining of ML pipelines on overlapping intervals of time-ordered data. As a result, they do not apply to deployment settings where new training data continuously arrives.

In multi-pipeline deployment settings, the work of van der Weide et al. [74] can be considered close to our work as it proposes a system for versioning the intermediate data in ML pipelines. However, the authors propose a brute-force approach, i.e., materialize every generated intermediate data and execute the pipelines one at a time.

TFX [18], as discussed earlier, is a system for deployment and retraining of ML pipelines. To optimize the retraining process, TFX materializes the last-level features of the deployed pipeline. In consecutive retrainings, the pipeline only processes the newly available training data and reuses any existing materialized data. However, TFX only supports the optimization of one pipeline at a time. Furthermore, TFX only materializes the last-level features of the pipeline. Therefore, in multi-pipeline deployment settings, TFX does not support inter-pipeline optimization. Moreover, TFX does not automatically address the problem of distribution change. Therefore, in the presence of a distribution change, users must manually invalidate older materialized data.

6 Additional Contributions

This chapter introduces additional contributions of the author, which are not covered in previous chapters. These contributions are made while working on the thesis and are closely related to the topics covered in the thesis.

1. Baunsgaard, Sebastian, Matthias Boehm, Ankit Chaudhary, **Behrouz Derakhshan**, Stefan Geißelsöder, Philipp M. Grulich et al. "ExDRa: Exploratory Data Science on Federated Raw Data." In Proceedings of the 2021 International Conference on Management of Data, pp. 2450-2463. 2021.

2. Ioannis Prapas, **Behrouz Derakhshan**, Alireza Rezaei Mahdiraji, and Volker Markl. "Continuous Training and Deployment of Deep Learning Models" Datenbank Spektrum (2021).

The paper *"ExDRa: Exploratory Data Science on Federated Raw Data"* introduces a system that addresses the requirements of data science workflows on raw data. The paper argues that data science projects deal with open-ended questions, which require exploratory analysis to design and refine ML pipelines. Another major challenge in data science projects is the limited access to data sources for solving the given problems. This is due to privacy laws that do not allow central data consolidation. The ExDRa system offers a federated runtime that can tackle federated data organization and federated linear algebra to enable exploratory analysis on federated datasets. A component of the ExDRa system is a model and artifact management system for storing intermediate data and models to enable meta-analysis as well as reuse of intermediates for speeding up the execution. This component of the system borrows ideas based on the work we described in Chapter 2.

The paper *"Continuous Training and Deployment of Deep Learning Models"* examines the applicability of our proposed solutions for optimizing the single-pipeline deployment workloads to neural network models and deep learning [119]. The paper shows that the *proactive training* approach can drastically decrease the training overhead of neural network models. Since proactive training updates the model frequently, transferring the updated model to the deployment environment incurs a large overhead. Deep neural network models are typically very large. Furthermore, when the deployment environment is outside of the training unit, the network transfer exacerbates the problem. In the paper, we utilize existing sparsification and compression techniques to solve the high transfer cost [120, 121, 122]. Combined with proactive training, every model update only transfers a small fraction of the gradients. In the paper, we show that applying proactive training and sparsification not only maintains the quality of the model when compared with existing retraining approaches but also improves the quality in some cases.

7 Conclusion

Machine learning applications are iterative in nature. To solve machine tasks, data scientists typically design several pipelines with different models and hyperparameters. Based on the performance, they then modify the existing pipelines and incrementally build a solution. This iterative nature results in redundant and overlapping data processing and model training operations that incur extra overhead. The goal of this thesis is to identify such redundancies and optimize the execution of ML applications by utilizing established database optimization techniques, i.e., view materialization and multi-query optimization.

To that end, in this thesis, we study the different ML workloads that are part of the end-to-end ML application life cycles. The result of the study is the identification of three common settings in ML applications life cycle, i.e., collaborative, single-pipeline deployment, and multi-pipeline deployment settings. The ML workloads in each setting exhibit similar computational redundancies. For each setting, we propose materialization and reuse optimizations to improve the performance of workload execution. Our solution to each workload category improves the performance by up to one order of magnitude when compared with the state-of-the-art solutions.

Future Work

This thesis lays the foundation for future work in several areas. In Chapter 2, we introduce Experiment Graph, which is the collection of all the executed workloads DAGs (directed acyclic graphs). Experiment Graph contains valuable information about the meta-data and hyperparameters of the feature engineering and model training operations. In future work, we plan to utilize this information to automatically construct ML pipelines and tune hyperparameters [123, 124, 125]; thus, fully or partially automating the process of designing ML pipelines.

In Chapter 4, we propose techniques for materializing and reusing artifacts in multi-pipeline settings, where the training schedule of the pipelines was known apriori. Another common approach in the deployment of ML pipelines is to trigger the retraining when the error rate of the deployed models falls below a threshold [19]. In future work, we plan to adapt our materialization algorithm to consider such cases by dynamically re-adjusting the benefit score of an artifact based on the current error rate of a model. Essentially, using the past performance of a model, the materialization algorithm can estimate when the pipeline will be retrained next and adjust the benefit score of the artifacts in the pipeline.

Lastly, in both Chapters 2 and 4, we represent workloads as DAGs. Our reuse procedure is lineage-based. This indicates that we reuse artifacts only when they have the same lineage, i.e., the ordered list of operators that resulted in the artifact. A future goal is to consider operator reordering, which enables the reuse procedure to consider artifacts with *equivalent plans* even if the lineages are different.

References

[1] Imene Mami and Zohra Bellahsene. “A survey of view selection methods”. In: *ACM SIGMOD Record* 41.1 (2012), pp. 20–29.

[2] Prasan Roy et al. “Efficient and extensible algorithms for multi query optimization”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 249–260.

[3] Nilesh N Dalvi et al. “Pipelining in multi-query optimization”. In: *Journal of Computer and System Sciences* 66.4 (2003), pp. 728–762.

[4] Timos K Sellis. “Multiple-query optimization”. In: *ACM Transactions on Database Systems (TODS)* 13.1 (1988), pp. 23–52.

[5] Ce Zhang, Arun Kumar, and Christopher Ré. “Materialization optimizations for feature selection workloads”. In: *ACM Transactions on Database Systems (TODS)* 41.1 (2016), pp. 1–32.

[6] Doris Xin et al. “Helix: Holistic optimization for accelerating iterative machine learning”. In: *Proceedings of the VLDB Endowment* 12.4 (2018), pp. 446–460.

[7] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[8] Sebastian Schelter et al. “On challenges in machine learning model management”. In: (2018).

[9] Nirmal Govind. *A/B Testing and Beyond: Improving the Netflix Streaming Experience with Experimentation and Data Science*. 2017. URL: `https://netflixtechblog.com/a-b-testing-and-beyond-improving-the-netflix-streaming-experience-with-experimentation-and-data-5b0ae9295bdf`.

[10] Kieran Kavanagh, David Nigenda, and Aakash Pydi. *A/B Testing ML models in production using Amazon SageMaker*. 2020. URL: `https://aws.amazon.com/blogs/machine-learning/a-b-testing-ml-models-in-production-using-amazon-sagemaker/`.

[11] Matthias Feurer et al. “Auto-sklearn: efficient and robust automated machine learning”. In: *Automated Machine Learning*. Springer, Cham, 2019, pp. 113–134.

[12] Daniel Crankshaw et al. “Clipper: A low-latency online prediction serving system”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[13] Thomas Kluyver et al. “Jupyter Notebooks – a publishing format for reproducible computational workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas.* Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[14] Kaggle. *Kaggle Data Science Platform.* San Francisco, United States, 2010. URL: `https://www.kaggle.com`.

[15] Google. *Google Colaboratory.* Seattle, WA, 2018. URL: `https://colab.research.google.com`.

[16] H Brendan McMahan 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.* 2013, pp. 1222–1230.

[17] J. Duchi, E. Hazan, and Y. Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[18] Denis Baylor 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.* 2017, pp. 1387–1395.

[19] Daniel Crankshaw et al. “The missing piece in complex analytics: Low latency, scalable model management and serving with velox”. In: *arXiv preprint arXiv:1409.3809* (2014).

[20] Justin Ma et al. “Identifying suspicious URLs: an application of large-scale online learning”. In: *Proceedings of the 26th annual international conference on machine learning.* 2009, pp. 681–688.

[21] Cedric Renggli et al. “Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment”. In: *Proceedings of Machine Learning and Systems.* Ed. by A. Talwalkar, V. Smith, and M. Zaharia. Vol. 1. 2019, pp. 322–333.

[22] Bojan Karlaš et al. “Building continuous integration services for machine learning”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* 2020, pp. 2407–2415.

[23] Google Cloud. *MLOps: Continuous delivery and automation pipelines in machine learning.* 2020. URL: `https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning`.

[24] Ihab F Ilyas and Xu Chu. *Data cleaning.* ACM, 2019.

[25] L. Bottou. “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of COMPSTAT’2010.* Springer, 2010, pp. 177–186.

[26] Home Credit Group. *Home Credit Default Risk.* 2019. URL: `https://www.kaggle.com/c/home-credit-default-risk/` (visited on 10/01/2019).

[27] OpenML. *Supervised Classification on credit-g (Task 31).* Oct. 2019. URL: `https://www.openml.org/t/31`.

[28] AzureML Team. “AzureML: Anatomy of a machine learning service”. In: *Conference on Predictive APIs and Apps.* 2016, pp. 1–13.

[29] Joaquin Vanschoren et al. "OpenML: networked science in machine learning". In: *ACM SIGKDD Explorations Newsletter* 15.2 (2014), pp. 49–60.

[30] Manasi Vartak et al. "ModelDB: A System for Machine Learning Model Management". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450342070. DOI: `10.1145/2939502.2939516`. URL: `https://doi.org/10.1145/2939502.2939516`.

[31] Joaquin Vanschoren et al. "Experiment databases". In: *Machine Learning* 87.2 (May 2012), pp. 127–158. ISSN: 1573-0565. DOI: `10.1007/s10994-011-5277-0`. URL: `https://doi.org/10.1007/s10994-011-5277-0`.

[32] Michelle Ufford et al. *Beyond Interactive: Noteook Innovation at Netflix*. Aug. 2018. URL: `https://medium.com/netflix-techblog/notebook-innovation-591ee3221233`.

[33] Will Koehrsen. *Kaggle Notebook, Start Here: A Gentle Introduction*. Oct. 2019. URL: `https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction`.

[34] Will Koehrsen. *Kaggle Notebook, Introduction to Manual Feature Engineering*. Oct. 2019. URL: `https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering`.

[35] Will Koehrsen. *Kaggle Notebook, Introduction to Manual Feature Engineering Part 2*. Oct. 2019. URL: `https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering-p2`.

[36] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[37] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[38] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[39] Manasi Vartak et al. "MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1285–1300. ISBN: 9781450347037. DOI: `10.1145/3183713.3196934`. URL: `https://doi.org/10.1145/3183713.3196934`.

[40] James Max Kanter and Kalyan Veeramachaneni. "Deep feature synthesis: Towards automating data science endeavors". In: *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2015, pp. 1–10.

[41] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[42] Souvik Bhattacherjee et al. “Principles of dataset versioning: Exploring the recreation/storage tradeoff”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1346–1357.

[43] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*. Vol. 5. Springer, 2007.

[44] Michael TM Emmerich and André H Deutz. “A tutorial on multiobjective optimization: fundamentals and evolutionary methods”. In: *Natural computing* 17.3 (2018), pp. 585–609.

[45] Carlos Roberto. *Kaggle Notebook, Start Here: A Gentle Introduction 312251*. Oct. 2019. URL: `https://www.kaggle.com/crldata/start-here-a-gentle-introduction-312251`.

[46] zhong xiao tao zhong xiao. *Kaggle Notebook, Begining with LightGBM*. Oct. 2019. URL: `https://www.kaggle.com/taozhongxiao/begining-with-lightgbm-in-detail`.

[47] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[48] Jon Kleinberg and Eva Tardos. *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.

[49] Jack Edmonds and Richard M. Karp. “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. In: *J. ACM* 19.2 (Apr. 1972), pp. 248–264. ISSN: 0004-5411. DOI: `10.1145/321694.321699`. URL: `https://doi.org/10.1145/321694.321699`.

[50] H. McMahan 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.

[51] X. Ling et al. “Model Ensemble for Click Prediction in Bing Search Ads”. In: *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee. 2017, pp. 689–698.

[52] T. Zhang. “Solving large scale linear prediction problems using stochastic gradient descent algorithms”. In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 116.

[53] L. Bottou, Y. Bengio, et al. “Convergence properties of the k-means algorithms”. In: *Advances in neural information processing systems* (1995), pp. 585–592.

[54] Y. Koren, R. Bell, C. Volinsky, et al. “Matrix factorization techniques for recommender systems”. In: *Computer* 42.8 (2009), pp. 30–37.

[55] J. Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[56] T. Schaul, S. Zhang, and Y. LeCun. “No more pesky learning rates.” In: *ICML (3)* 28 (2013), pp. 343–351.

[57] N. Qian. “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1 (1999), pp. 145–151.

[58] D. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[59] T. Tieleman and G. Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[60] M. Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

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

[62] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Sequential model-based optimization for general algorithm configuration”. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2011, pp. 507–523.

[63] K. Yu et al. “Scalable and accurate online feature selection for big data”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11.2 (2016), p. 16.

[64] I. Tsamardinos et al. “A greedy feature selection algorithm for Big Data of high dimensionality”. In: *Machine Learning* (2018), pp. 1–54.

[65] Y. Park et al. “BlinkML: Approximate Machine Learning with Probabilistic Guarantees”. In: (2018).

[66] Z. Sun. “Arithmetic theory of harmonic numbers”. In: *Proceedings of the American Mathematical Society* 140.2 (2012), pp. 415–428.

[67] A. Gepperth and B. Hammer. “Incremental learning algorithms and applications”. In: *European Symposium on Artificial Neural Networks (ESANN)*. 2016.

[68] P. Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Data Engineering* (2015), p. 28.

[69] T. Akidau et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.

[70] O. 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].

[71] K. Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE. 2010, pp. 1–10.

[72] P. Dawid. “Present position and potential developments: Some personal views: Statistical theory: The prequential approach”. In: *Journal of the Royal Statistical Society. Series A (General)* (1984), pp. 278–292.

[73] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. Manning, 2013.

[74] Tom van der Weide et al. “Versioning for end-to-end machine learning pipelines”. In: *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*. 2017, pp. 1–9.

[75] Xinran He et al. "Practical lessons from predicting clicks on ads at facebook". In: *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising.* 2014, pp. 1–9.

[76] Behrouz Derakhshan et al. "Continuous Deployment of Machine Learning Pipelines." In: *EDBT.* 2019, pp. 397–408.

[77] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[78] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[79] Michael Greenwald and Sanjeev Khanna. "Space-efficient online computation of quantile summaries". In: *ACM SIGMOD Record* 30.2 (2001), pp. 58–66.

[80] Jim Gray et al. "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals". In: *Data mining and knowledge discovery* 1.1 (1997), pp. 29–53.

[81] Cheng-Hao Tsai, Chieh-Yen Lin, and Chih-Jen Lin. "Incremental and decremental training for linear classification". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining.* 2014, pp. 343–352.

[82] Jordan T Ash and Ryan P Adams. "On the difficulty of warm-starting neural network training". In: *arXiv preprint arXiv:1910.08475* (2019).

[83] "Kolmogorov–Smirnov Test". In: *The Concise Encyclopedia of Statistics.* New York, NY: Springer New York, 2008, pp. 283–287. ISBN: 978-0-387-32833-1. DOI: `10.1007/978-0-387-32833-1_214`. URL: `https://doi.org/10.1007/978-0-387-32833-1_214`.

[84] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: `10.1145/2934664`. URL: `https://doi.org/10.1145/2934664`.

[85] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[86] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12).* 2012, pp. 15–28.

[87] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nature communications* 5.1 (2014), pp. 1–9.

[88] NYC Taxi and Limousine Commission (TLC). *NYC Taxi and Limousine Commission (TLC) Trip Record Data.* 2021. URL: `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

[89] Kaggle. *New York City Taxi Trip Duration Competition.* 2020. URL: `https://www.kaggle.com/c/nyc-taxi-trip-duration/`.

[90] Jeffrey Chung. *NYC Taxi Trip - Public.* 2020. URL: `kaggle.com/jeffreycbw/nyc-taxi-trip-public-0-37399-private-0-37206`.

[91] Quentin Monmousseau. *ML Workflow.* 2019. URL: kaggle.com/quentinmonmousseau/ml-workflow-lightgbm-0-37-randomforest-0-39.

[92] Ahmed Mazen and Motaz Saad. *NYC Taxi.* 2021. URL: kaggle.com/ahmedmurad1990/nyc-taxi.

[93] A. P. Dawid. "Present Position and Potential Developments: Some Personal Views: Statistical Theory: The Prequential Approach". In: *Journal of the Royal Statistical Society. Series A (General)* 147.2 (1984), pp. 278–292. ISSN: 00359238. URL: http://www.jstor.org/stable/2981683.

[94] Google. *Google AI Platform.* California, USA, 2018. URL: https://cloud.google.com/ai-platform/.

[95] Matei Zaharia et al. "Accelerating the Machine Learning Lifecycle with MLflow." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 39–45.

[96] Sebastian Schelter et al. "Automatically tracking metadata and provenance of machine learning experiments". In: *Machine Learning Systems workshop at NIPS.* 2017.

[97] Anant Bhardwaj et al. "Datahub: Collaborative data science & dataset version management at scale". In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2015.

[98] Rolando Garcia et al. "Context: The missing piece in the machine learning lifecycle". In: *KDD CMI Workshop.* Vol. 114. 2018.

[99] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2017.

[100] Hui Miao and Amol Deshpande. "ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows". In: *IEEE Data Eng. Bull.* 41 (2018), pp. 26–38.

[101] Raul Castro Fernandez et al. "Aurum: A data discovery system". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE. 2018, pp. 1001–1012.

[102] Zack Ives et al. "Dataset Relationship Management." In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2019.

[103] M. Akdere et al. "The Case for Predictive Database Systems: Opportunities and Challenges." In: *CIDR.* 2011, pp. 167–174.

[104] D. Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *arXiv preprint arXiv:1612.03079* (2016).

[105] Pradeep Kumar Gunda et al. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 75–88.

[106] Iman Elghandour and Ashraf Aboulnaga. "ReStore: reusing results of MapReduce jobs". In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 586–597.

[107] Tomasz Nykiel et al. "MRShare: sharing across multiple queries in MapReduce". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 494–505.

[108] Guoping Wang and Chee-Yong Chan. "Multi-query optimization in mapreduce framework". In: *Proceedings of the VLDB Endowment* 7.3 (2013), pp. 145–156.

[109] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". In: *Communications of the ACM* 53.1 (2010), pp. 72–77.

[110] Yuan Yu Michael Isard Dennis Fetterly et al. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language". In: *Proc. LSDS-IR* 8 (2009).

[111] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. "AJoin: ad-hoc stream joins at scale". In: *Proceedings of the VLDB Endowment* 13.4 (2019), pp. 435–448.

[112] Jonas Traub et al. "Efficient Window Aggregation with General Stream Slicing". In: *EDBT*. 2019, pp. 97–108.

[113] Kanat Tangwongsan et al. "General incremental sliding-window aggregation". In: *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 702–713.

[114] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. "On-the-fly sharing for streamed aggregation". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006, pp. 623–634.

[115] Evan R Sparks et al. "Keystoneml: Optimizing pipelines for large-scale advanced analytics". In: *2017 IEEE 33rd international conference on data engineering (ICDE)*. IEEE. 2017, pp. 535–546.

[116] Sona Hasani et al. "Efficient construction of approximate ad-hoc ML models through materialization and reuse". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1468–1481.

[117] Sona Hasani et al. "ApproxML: efficient approximate ad-hoc ML models through materialization and reuse". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1906–1909.

[118] Jaeho Shin et al. "Incremental knowledge base construction using deepdive". In: *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*. Vol. 8. 11. NIH Public Access. 2015, p. 1310.

[119] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[120] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. "Sparsified SGD with memory". In: *arXiv preprint arXiv:1809.07599* (2018).

[121] Alham Fikri Aji and Kenneth Heafield. "Sparse communication for distributed gradient descent". In: *arXiv preprint arXiv:1704.05021* (2017).

[122] Yujun Lin et al. "Deep gradient compression: Reducing the communication bandwidth for distributed training". In: *arXiv preprint arXiv:1712.01887* (2017).

[123] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2755–2763.

[124] Chris Thornton et al. “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms”. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2013, pp. 847–855.

[125] Zeyuan Shang et al. “Democratizing Data Science through Interactive Curation of ML Pipelines”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1171–1188. ISBN: 9781450356435. DOI: `10.1145/3299869.3319863`. URL: `https://doi.org/10.1145/3299869.3319863`.