Optimizing Machine Learning Workloads in Collaborative Environments

Behrouz Derakhshan
DFKI GmbH
behrouz.derakhshan@dfki.de

Alireza Rezaei Mahdiraji
DFKI GmbH
alireza.rm@dfki.de

Ziawasch Abedjan
TU Berlin
abedjan@tu-berlin.de

Tilmann Rabl*
Hasso Plattner Institute
University of Potsdam
tilmann.rabl@hpi.de

Volker Markl
DFKI GmbH
TU Berlin
volker.markl@tu-berlin.de

ABSTRACT

Effective collaboration among data scientists results in high-quality and efficient machine learning (ML) workloads. In a collaborative environment, such as Kaggle or Google Colabratory, users typically re-execute or modify published scripts to recreate or improve the result. This introduces many redundant data processing and model training operations. 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 paper, we present a system to optimize the execution of ML workloads in collaborative environments by reusing previously performed operations and their results. We utilize a so-called Experiment Graph (EG) to store 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 based on their likelihood of future reuse. 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 scales for the high number of incoming ML workloads in collaborative environments. Our experiments show that we improve the run-time by one order of magnitude for repeated execution of the workloads and 50% for the execution of modified workloads in collaborative environments.

ACM Reference Format:
Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389715

1 INTRODUCTION

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 [37] and Kaggle [20] provide a collaborative environment where users share their scripts and results using Jupyter notebooks [23]. Other platforms such as OpenML [41] and ModelDB [43] enable collaboration by storing ML pipelines, hyperparameters, models, and evaluation results in experiment databases [40].

The collaborative platforms 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, the collaborative platforms improve the execution of future workloads by skipping redundant operations. However, the existing collaborative platforms lack automatic management of the stored artifacts and require the users to manually search through the artifacts and incorporate them into their workloads. In the current collaborative environments, we

*Work was partially done while author was at TU Berlin.

SIGMOD'20, June 14 - June 19, 2020, Portland, Oregon, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389715

identify two challenges that prohibit the 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 generate up to 125 GB of artifacts. Second, ML workloads have a complex structure; thus, automatically finding artifacts for reuse is challenging.

We propose a solution for optimizing the execution of ML workloads, which addresses these two challenges. Our solution stores the artifacts with a high likelihood of reappearing in the 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 of 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 the *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. (1) We propose a system to optimize the execution of ML workloads in collaborative environments. (2) We present Experiment Graph, a collection of the artifacts and operations of the ML workloads. (3) 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. (4) We propose a linear-time reuse algorithm for generating optimal execution plans for the ML workloads.

The rest of this paper is organized as follows. In Section 2, we provide some background information. We introduce our collaborative workload optimizer in Section 3. In Section 4, we discuss our data model and programming API. In Sections 5 and 6, we introduce the materialization and reuse algorithms. In Section 7, we present our evaluations. In Section 8, we discuss the related work. Finally, we conclude this work in Section 9.

2 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 [23] to write and execute their scripts in isolated containers [29] within the server itself [15, 20, 39]. 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.

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 [24–26]. 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

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

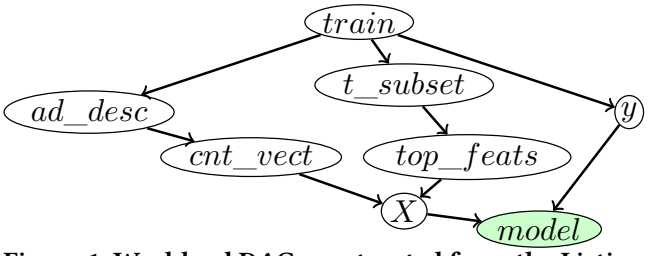

Figure 1: Workload DAG constructed from the Listing 1. The highlighted node shows a terminal vertex.

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.

3 COLLABORATIVE ML WORKLOAD OPTIMIZATIONS

In this section, we present our collaborative ML workload optimization system. Figure 2 shows the architecture of our system, which comprises of 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 the Experiment Graph and store the artifacts of the workload DAG (Step 5). This architecture enables us to integrate our system into the existing 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.

3.1 Client Components

ML Script and Parser. We design an extensible DSL, which enables integration with Python data analysis and ML packages, such as Pandas [28] and scikit-learn [5]. After invoking a script, the parser generates a DAG. Listing 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 1 shows the workload DAG constructed

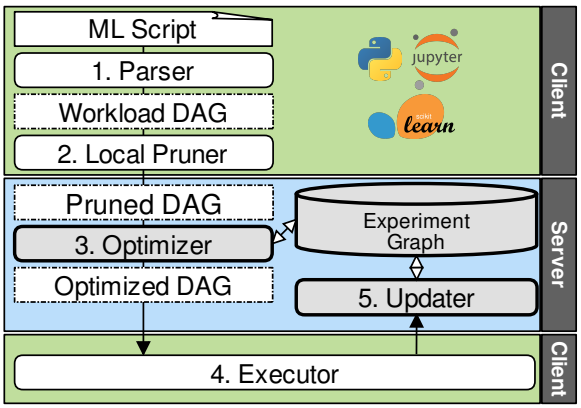

Figure 2: Collaborative workload optimizer system

of the code in Listing 1. In the Figure, the terminal vertex is the result of the print statement on Line 16 in Listing 1.

```
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 1: Example script

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 the 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 1, if *t_subset* is computed, the local pruner marks 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.

3.2 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 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.

3.3 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.

4 REPRESENTATION AND PROGRAMMING INTERFACE

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

4.1 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 [28], (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 [47] handles conditional control flows.

4.2 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 1 (Section 3) 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.

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. 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.

Listing 2 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, 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 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, 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. However, users must specify whether the training operation can be warmstarted or not.

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 [21], and ML frameworks, such as TensorFlow [1]. 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.

```
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: Defining and using a new operation

5 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, it may be less costly to recompute an artifact from scratch. In this section, we introduce two algorithms for materializing the artifacts with a high likelihood of future reuse while ensuring the storage does not surpass the recomputation cost. The first algorithm (Section 5.2) 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 5.3) 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 the 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.

5.1 Materialization Problem Formulation

Existing work proposes algorithms for the efficient storage of dataset versions and their storage and recomputation trade-off [4]. 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:

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

Existing work proves that minimizing the recreation cost alone is an NP-Hard problem [4]. While there are different approximate strategies for solving multi-objective optimization problems [6], they are time-consuming, which renders them inappropriate to our setting, where new workloads are constantly executed. As a result, existing solutions to multi-objective optimization problems are not suitable for artifact materializations of EG.

5.2 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 [9], 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

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  M := ∅;                          // materialized set
3  PQ := empty priority queue;
4  for v ← V do
5      if v.mat = 0 then
6          v.utility := 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 ≤ B then
11         M := M ∪ v;
12         S := S + v.s;
13 return 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 equation (Equation 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:

$$\mathcal{U}(v) := \begin{cases} 0, & \text{if } C_l(v) \geq C_r(v) \\ \alpha p'(v) + (1-\alpha) r'_{cs}(v), & \text{otherwise} \end{cases} \quad (2)$$

, 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 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 the Experiment Graph. Thus, the complexity of the materialization algorithm is $O(|V|)$ 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 $O(|W| + |M|)$, where $|W|$ is the number of vertices in the new workload DAG and $|M|$ is the number of the materialized vertices.

5.3 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 (Artifact-Materialization). 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.

6 REUSE AND WARMSTARTING

Our collaborative 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.

6.1 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 5). 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 the Experiment Graph into the workload DAG. The backward-pass prunes any unnecessary materialized vertices before transferring the optimized DAG to the client.

Algorithm 2: Forward-pass

Input: G_W workload DAG, G_E experiment graph
Output: $\mathcal{R}$ set of vertices for reuse

1 **for** $s \in sources(G_W)$ **do**
2 $recreation\_cost[s] := 0$;
3 $\mathcal{R} := \emptyset$;
4 **for** $v \leftarrow topological\_order(G_W)$ **do**
5 **if** v *computed in* G_W **then**
6 $recreation\_cost[v] := 0$;
7 **else**
8 $p\_costs := \sum_{p \in parents(v)} recreation\_cost[p]$;
9 $execution\_cost := C_i(v) + p\_costs$;
10 **if** $C_l(v) < execution\_cost$ **then**
11 $recreation\_cost[v] := C_l(v)$;
12 $\mathcal{R} := \mathcal{R} \cup v$;
13 **else**
14 $recreation\_cost[v] := execution\_cost$;
15 return $\mathcal{R}$;

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* dictionary. The client always loads the source artifacts completely. Therefore, we set their recreation cost to 0 (Lines 1 and 2). Then, we visit every vertex in their 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 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, 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.

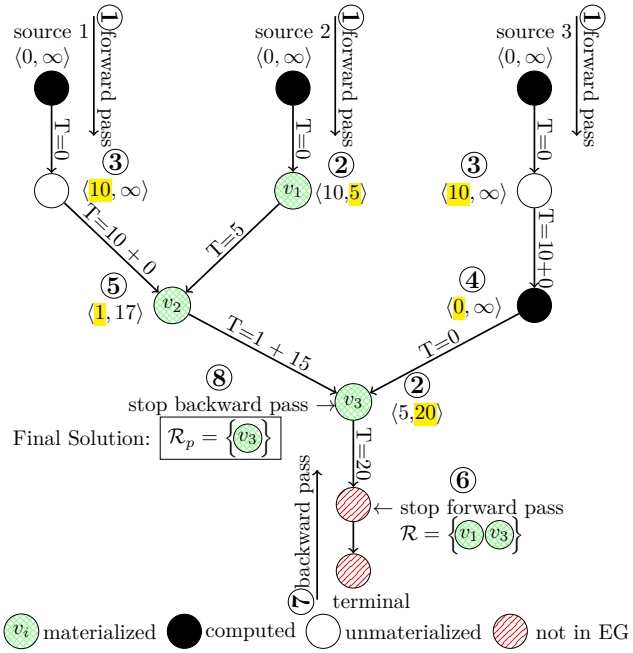

Figure 3: Reuse Algorithm Example. Each vertex has the label $\langle C_i(v), C_l(v)\rangle$. T is the recreation cost.

Figure 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 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 $O(|V|)$. A linear-time algorithm can scale to a large number of workloads, which is typical in collaborative environments such as Kaggle.

6.2 Warmstarting

Many model training operations include different hyperparameters, which impact the training process. Two training operations on the same dataset with different hyperparameters may 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 shown to decrease the total training time in some scenarios [2]. 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.

7 EVALUATION

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

7.1 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 [17]. We run every experiment 3 times and report the average.

Baseline and other System. We compare our system with a naive baseline, i.e., executing all the workloads without any optimization, and Helix [45]. 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 [45]). To find the optimal reuse plan, Helix reduces the workload DAG into an instance of the project selection problem (PSP) and solve it via the Max-Flow algorithm [22]. 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 [7], which runs in polynomial time ($O(|V|.|E|^2)$).

Kaggle workloads. In the Kaggle workloads, we recreate the use case in Section 2. We use eight workloads, which generate 130 GB of artifacts. There are five real and three custom workloads. Table 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 [31]. 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 $\alpha = 0.5$.

7.2 End-to-end Optimization

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

Figure 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

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 [26].	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 [24].	406	25
3	A real Kaggle script. It is similar to Workload 2, with the resulting preprocessed datasets having more features [25].	146	83.5
4	A real Kaggle script that modifies Workload 1 and trains a gradient boosted tree with a different set of hyperparameters [32].	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 [36].	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

Table 1: Description of Kaggle workloads. N is number of the artifacts and S is the total size of the artifacts in GB.

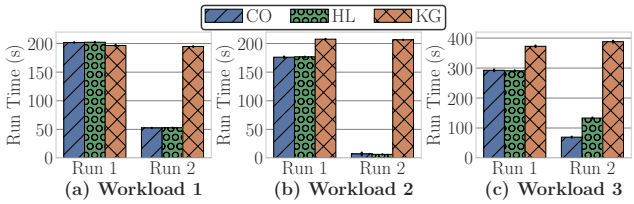

Figure 4: Repeated executions of Kaggle workloads

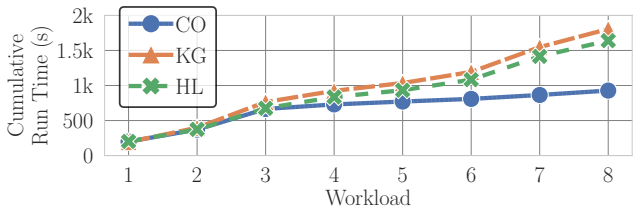

Figure 5: Execution of Kaggle workloads in sequence

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 the first run. Both Workloads 2 and 3 contain many redundant operations. The local pruning step of our optimizer identifies the redundancies and only execute 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 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 root node 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 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 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.

7.3 Materialization

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

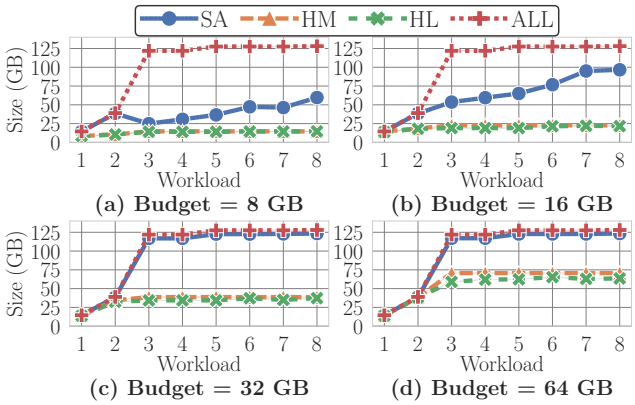

Figure 6: Real size of the materialized artifact

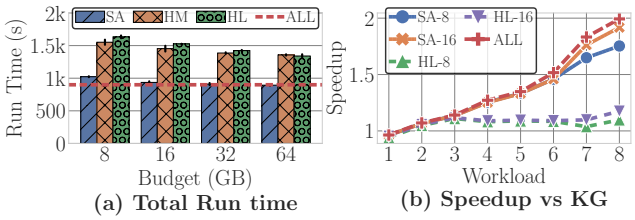

Figure 7: Total run-time and speedup (vs baseline)

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. Figures 6(a)-(d) show 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 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 6(a) 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 7(a) shows the total run-time of different materialization algorithms and budgets. ALL represents the scenario where all the artifacts are 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 ($\leq 16GB$), where HL performs 20% and 90% worse than HM and SA, respectively. For larger budgets, HL has similar performance to HM, since a larger fraction of all the artifacts is materialized.

In Figure 7(b), 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 in the collaborative environment. Such a scenario is common in collaborative environments, where users constantly attempt to improve the best current model. We use the OpenML workloads for the model-benchmarking scenario. The implementation of the 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 8(a) 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 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 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.

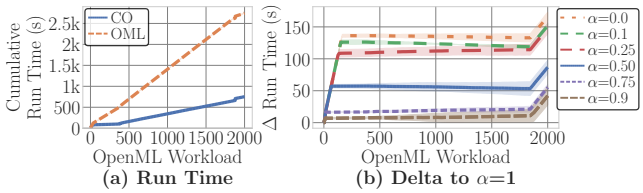

Figure 8: Effect of quality-based materialization

In Figure 8(b), 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 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.

7.4 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 9(a) and (b) 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 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 1). As a result, there are not many reuse opportunities until Workload 7, which is a modified version of Workload 3.

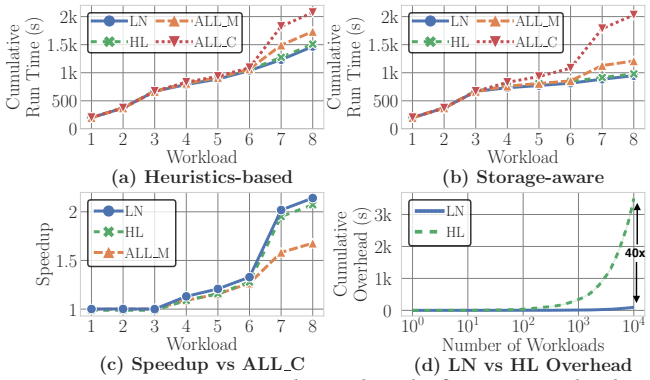

Figure 9: Run-time and overhead of reuse methods

The storage-aware materialization (Figure 9(b)) has better budget utilization and materializes some of the artifacts of the 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 9(c). 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 of 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. 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 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 outdegree more than 1 represents a scenario were 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,

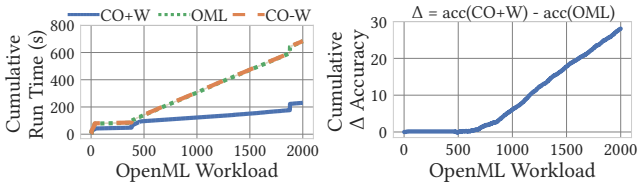

Figure 10: Warmstarting of OpenML workloads

which represents many of the Kaggle workloads. Then, for every node, we sample its attributes from the distributions of attributes of the real workloads.

Figure 9(d) 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 environment, where hundreds of users are executing workloads, a large reuse overhead leads to slower response time and may cause a bottleneck during the optimization.

7.5 Warmstarting

In this experiment, we evaluate the effect of our warmstarting method. Figure 10 shows the effect of warmstarting on run-time and accuracy for the OpenML workloads. In Figure 10(a), 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 10(b), 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. The figure 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.

8 RELATED WORK

Our system lies at the intersection of 3 categories of existing work. The first category consists of data science platforms that enable collaboration between users. The second category is data management and provenance systems that capture the relationship between data artifacts. The last category contains ML and database systems that optimize workloads through materialization and reuse.

Collaborative Data Science Platforms. Cloud-based systems, such AzureML [37], Google's AI platform [14], Kaggle [20], 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. However, none of these systems manage the generated ML artifacts and do not utilize them to optimize the execution of the workloads. Our system manages the ML artifacts and offers reuse and warmstarting methods to decrease the execution time of the future workloads.

OpenML [41], ModelDB [43], and MLflow [46] are platforms that store ML artifacts, such as models and intermediate datasets, in a database [33, 40]. These platforms provide APIs for users to query the details of the ML artifacts. Contrary to our systems, none of these platforms offer automatic materialization and reuse of the ML artifacts.

Data Management and Provenance. DataHub [3, 4], Context [13], Ground [18], ProvDB [30], Aurum [10], and JuNEAU [19] are data management and provenance systems that efficiently store fine-grained lineage information about the data artifacts and operations. We design our Experiment Graph 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 workloads. Our materialization algorithm extends the materialization approach of Bhattacherjee et al. [4] to tailor it to ML workloads by considering the quality of the model artifacts.

Materialization and Reuse in ML and Databases. View selection is a long-studied problem in databases, which concerns itself with finding an appropriate set of views to materialize to speed up the execution of queries [27]. Several ML systems utilize such techniques to optimize the execution of workloads. Helix [44, 45], DeepDive [48], Columbus [49], KeystoneML [35], and Mistique [42] are ML systems which optimize workloads by materializing intermediate data for reuse. These systems have three fundamental differences with our system. 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. 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. Second, the materialization decisions in these systems only utilize run-time and size and do not take into account the model quality. Third, our system operates in a collaborative and multi-tenant environment. Whereas, the scope of optimization in these systems, except for Mistique, is limited to a single session. However, Mistique is a model diagnosis tool, which enables users to query intermediate artifacts from an artifact store efficiently. Whereas, we focus on generating optimal execution plans for future workloads by reusing the artifacts in EG. Nectar [16] and ReStore [8] offer materialization and reuse of intermediate data generated in DryadLINQ [11] and MapReduce jobs. However, Both of these systems only support simple data processing pipelines and do not offer any optimizations for ML workloads.

9 CONCLUSIONS

We present a system for optimizing machine learning workloads in collaborative environments. We propose a graph representation of the artifacts of ML workloads, which we refer to as the Experiment Graph. Using EG, we offer materialization and reuse algorithms. We propose two materialization algorithms. The heuristics-based algorithm stores artifacts of the graph 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.

We show that our collaborative optimizer improves the execution time of ML 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.

Future work. EG 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 [12, 34, 38]; thus, fully or partially automating the process of designing ML pipelines.

ACKNOWLEDGMENTS

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A) and German Federal Ministry for Economic Affairs and Energy, Project "ExDRa" (01MD19002B).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.

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

[3] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshp, Aaron J. Elmore, Samuel Madden, and Aditya Parameswaran. 2015. Datahub: Collaborative data science & dataset version management at scale. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[4] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1346–1357.

[5] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, et al. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

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

[7] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (April 1972), 248–264. https://doi.org/10.1145/321694.321699

[8] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: reusing results of MapReduce jobs. *Proceedings of the VLDB Endowment* 5, 6 (2012), 586–597.

[9] Michael TM Emmerich and André H Deutz. 2018. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural computing* 17, 3 (2018), 585–609.

[10] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[11] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR* 8 (2009).

[12] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. MIT Press, Cambridge, MA, USA, 2755–2763.

[13] Rolando Garcia, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E Gonzalez, and Joseph M Hellerstein. 2018. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, Vol. 114.

[14] Google. 2018. Google AI Platform. https://cloud.google.com/ai-platform/

[15] Google. 2018. Google Colaboratory. https://colab.research.google.com

[16] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, USA, 75–88.

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

[18] Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, et al. 2017. Ground: A Data Context Service.. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[19] Zack Ives, Yi Zhang, Soonbo Han, and Nan Zheng. 2019. Dataset Relationship Management.. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[20] Kaggle. 2010. Kaggle Data Science Platform. https://www.kaggle.com

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

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

[23] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, et al. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[24] Will Koehrsen. 2019. Kaggle Notebook, Introduction to Manual Feature Engineering. Retrieved October 11, 2019 from https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering

[25] Will Koehrsen. 2019. Kaggle Notebook, Introduction to Manual Feature Engineering Part 2. Retrieved October 11, 2019 from https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering-p2

[26] Will Koehrsen. 2019. Kaggle Notebook, Start Here: A Gentle Introduction. Retrieved October 11, 2019 from https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction

[27] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD Rec.* 41, 1 (April 2012), 20–29. https://doi.org/10.1145/2206869.2206874

[28] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[29] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

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

[31] OpenML. 2019. Supervised Classification on credit-g (Task 31). Retrieved October 11, 2019 from https://www.openml.org/t/31

[32] Carlos Roberto. 2019. Kaggle Notebook, Start Here: A Gentle Introduction 312251. Retrieved October 11, 2019 from https://www.kaggle.com/crldata/start-here-a-gentle-introduction-312251

[33] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*.

[34] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1171–1188.

https://doi.org/10.1145/3299869.3319863

[35] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd international conference on data engineering (ICDE)*. IEEE, 535–546.

[36] zhong xiao tao. 2019. Kaggle Notebook, Begining with LightGBM. Retrieved October 11, 2019 from https://www.kaggle.com/taozhongxiao/begining-with-lightgbm-in-detail

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

[38] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. 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, 847–855.

[39] Michelle Ufford, M Pacer, Matthew Seal, and Kyle Kelley. 2018. Beyond Interactive: Noteook Innovation at Netflix. Retrieved October 4, 2019 from https://medium.com/netflix-techblog/notebook-innovation-591ee3221233

[40] Joaquin Vanschoren, Hendrik Blockeel, Bernhard Pfahringer, and Geoffrey Holmes. 2012. Experiment databases. *Machine Learning* 87, 2 (01 May 2012), 127–158. https://doi.org/10.1007/s10994-011-5277-0

[41] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. 2014. OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter* 15, 2 (2014), 49–60.

[42] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. 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)*. Association for Computing Machinery, New York, NY, USA, 1285–1300. https://doi.org/10.1145/3183713.3196934

[43] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 14, 3 pages. https://doi.org/10.1145/2939502.2939516

[44] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. 2018. Helix: accelerating human-in-the-loop machine learning. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1958–1961.

[45] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment* 12, 4 (2018), 446–460.

[46] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, USA, 10.

[48] Ce Zhang. 2015. DeepDive: a data management system for automatic knowledge base construction. *University of Wisconsin-Madison, Madison, Wisconsin* (2015).

[49] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization Optimizations for Feature Selection Workloads. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/2588555.2593678