Optimizing Machine Learning Workloads in Collaborative Data Science Platforms

Anonymous Author(s)

ABSTRACT

Collaborative data science platforms, such as Kaggle and OpenML, bring a new paradigm for solving machine learning tasks. Users write and execute their machine learning workloads and publish them on the platforms. The platforms store the workloads either in a relational database using a predefined schema (e.g., OpenML) or as scripts which contain the actual analysis and training code (e.g., Kaggle). This results in many redundant data processing and model training operations, as users tend to utilize published workloads to improve their solutions. However, current collaborative platforms cannot detect and exploit such redundancies in the workloads as they execute them in isolation.

In this paper, we present a system to optimize the execution of ML workloads in collaborative data science platforms. We utilize a graph to store the artifacts, i.e., raw and intermediate data or machine learning models, and operations of machine learning workloads as vertices and edges, which we refer to as the experiment graph. As more workloads are executed, the combined size of the artifacts grows rapidly, which renders their storage infeasible. To alleviate this problem, we propose two algorithms for materializing the artifacts with high likelihoods of future reuse. The algorithms consider several metrics, such as access frequency, size of the artifact, and quality of machine learning models to decide what artifacts to materialize. Second, using the experiment graph, we propose a novel approach to reuse artifacts and warmstart model training operations in future workloads.

KEYWORDS

Collaborative Data Science; Machine Learning Pipeline Management

1 INTRODUCTION

Machine learning is the process of analyzing training datasets to extract features and build machine learning models to solve different tasks, such as labeling images based on image content and detecting fraudulent credit card and bank transactions. To solve machine learning tasks, a data scientist designs and executes a machine learning workload consisting of a set of exploratory data transformation steps and one or multiple model building steps. Many of these workloads are executed in an interactive approach in notebook environments, where users can examine the result of every operation.

Recent collaborative data science platforms facilitate workload sharing, which enables data scientists to work together to write more optimized workloads and train better machine learning models. There are two categories of collaborative data science platforms. In the first category, machine learning workloads are scripts, such as Python or R scripts, or interactive Jupyter notebooks [9]. Kaggle [8] and Google Colabratory [7] are two popular examples of the first category, where users publicly publish their scripts. In the second category, the workloads are represented by machine learning pipelines comprising of a sequence of data preprocessing operations and model training operation. OpenML [15], ModelDB [17], and ProvDB [11] are examples of the second category, which are referred to as experiment databases [14]. Experiment databases typically store the components of the pipeline and the resulting model in a database.

Currently, collaborative data science platforms act as repositories of machine learning artifacts and execution platforms for machine learning workloads, i.e., scripts or pipelines. The platforms ignore the stored artifacts and operations, which provide valuable knowledge about the past workloads and as a result, miss optimization opportunities. By exploiting this knowledge, the platforms can improve the performance of future workloads by skipping redundant operations and by training more accurate machine learning models using the past trained models. There are two main challenges which collaborative platforms must address to take advantage of the knowledge. First, the amount and size of the artifacts which are generated in the workloads are large and storing everything is not feasible. Therefore, only artifacts which have a probability of reappearing in future workloads should be stored. Second, the platforms must organize the stored artifacts and quickly find them for reuse in future workloads, otherwise, the optimization overhead is too large.

We propose a solution which addresses these two challenges. We model a workload (script or pipeline) with a graph, where vertices represent the artifacts and edges represent the operations in the workload. Each artifact is uniquely identified using the sequence of operations that generated it. Typically, an artifact comprises of meta-data, such as the column names a dataframe or type and hyperparameters of a model, and underlying content, such as the actual data inside a dataframe or weight vector of a machine learning model. After a workload is executed, we store the graph. We refer to the collection of all the workload graphs as the

Experiment Graph. The Experiment Graph is itself a graph which contains the union of all the vertices and edges of the workload graphs. The Experiment Graph is then made available to all the users of the collaborative data science platform. The size of the meta-data of the artifacts and the operations is small and can be stored inside the Experiment Graph. However, the size of the underlying content of the artifacts, i.e., datasets and model weight vectors, is large. We propose the *Artifact Materialization* algorithm which receives the Experiment Graph and a storage budget as inputs and decides the underlying content of what artifacts to store. Our materialization algorithm utilizes several metrics such as the size, access frequency, operation run-time, and the score of the machine learning models to decide what are artifacts to store. Using the materialized Experiment Graph, we propose efficient reuse and model warmstarting methods. The process of reuse and warmstarting is as follows. For every artifact in a new workload, we search for the artifact, using its unique id, in the Experiment Graph. We reuse an artifact instead of executing the operations in the workload when the following two conditions are satisfied. The Experiment Graph contains the artifact and the underlying content of the artifact is materialized. When the artifact in question is a machine learning model, due to the stochasticity of some model training operations, we cannot always reuse an existing model from the Experiment Graph. Instead, we warmstart the training operation of the workload with a model artifact from the Experiment Graph. Depending on the number of the artifacts and where the experiment graph is stored, i.e., in an in-memory database, on disk, or in a remote database, a brute-force search creates a large overhead. In our reuse and warmstarting methods, we utilize early stopping and pruning techniques as well as heuristics to speed up the search and decrease the optimization overhead.

In summary, we make the following contributions:

- We propose a system for optimizing the execution of machine learning workloads in collaborative environments.
- We present Experiment Graph, a graph representation of the collection of the artifacts and operations the machine learning workloads.
- We propose an algorithm for materializing the artifacts in the Experiment Graph under limited storage capacity.
- We propose efficient reuse and warmstarting methods for new workloads.

The rest of this document is organized as follows. In Section 2, we provide some background information and show an example. We introduce our proposed collaborative workload optimizer system in Section 3. In Sections 4 and 5, we introduce the artifacts materialization algorithms, reuse strategy, and the warmstarting technique. In Section 6, we show the result of our evaluations. In Section 2, we discuss the related work and finally, we conclude this work in Section 8.

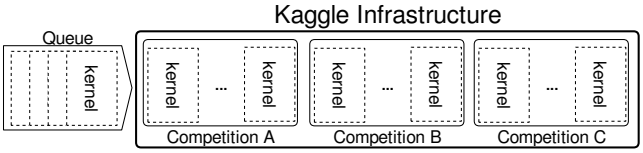


Figure 1: Kaggle Infrastructure

2 BACKGROUND

In this section, we first provide the workflow of a collaborative data science platform, which we use throughout the paper and outline the challenges and inefficiencies of the platform. Next, we describe experiment databases and how they can improve the execution of machine learning workloads.

2.1 Motivating Example

Kaggle is a data science competition platform which enables organizations to host data science challenges. Users participate in the challenges, write solutions, and submit these solutions to the platform. To arrive at their final solutions, participants utilize the infrastructure provided by Kaggle to write data analysis and machine learning workloads either in the form of R and Python scripts (a long running workload) or Jupyter notebooks (interactive workloads) and execute them on the Kaggle's platform. Users can also make their workloads publicly available to other users. As a result, many Kaggle users work together to find high-quality solutions.

Kaggle utilizes docker containers to provide isolated computational environments called Kaggle kernels and groups these kernels by competitions. Figure 1 shows the infrastructure of Kaggle. Each kernel has limited CPU, GPU, disk space, and memory (i.e., 4 CPU cores, 16 GB of RAM, 5 GB of disk space, and a maximum of 9 hours of execution time. GPU kernels have 2 CPU cores and 13 GB of RAM[1]). In busy times, this results in users to be placed in queues (especially for GPU-enabled machines) until resources become available.

Every user who participates in a Kaggle competition has the same goal, which is to solve the task described by the competition organizer. Typically, the task is to design a machine learning workload containing a series of exploratory data analysis steps to preprocess one or multiple raw datasets provided by the competition organizer, followed by a model training step to train a machine learning model, which aims to maximize a quality metric on an evaluation dataset. For

[1]https://www.kaggle.com/docs/kernels

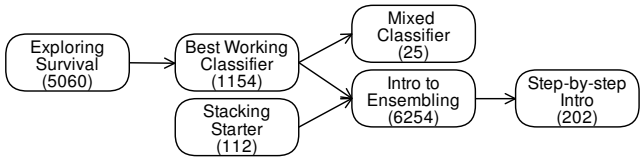


Figure 2: The fork hierarchy of some of the popular kernels in Kaggle's Titanic competition

example, in the *Titanic: Machine Learning from Disaster* competition in Kaggle[2], the task is to create a machine learning pipeline and train a classification model on the Titanic training dataset that can predict if a traveler survived the Titanic disaster, with the goal of maximizing the prediction accuracy on a separate test dataset. When solving the same tasks, users tend to utilize the same type of operations. Figure 2 shows the most popular kernels and their relationship for the Titanic competition. For example, in the kernel **Best Working Classifier**, the author explicitly cites the kernel **Exploring Survival** as his/her inspiration. The numbers show how many times other users copy a kernel into their workspace. The most popular kernels for the Titanic competition have been copied a total of 44,434 times. This demonstrates that many of the executed workloads share exact or similar operations. Since Kaggle is using isolated docker containers for executing user workloads, they cannot detect similar operations and re-execute the operations multiple times. Moreover, while users can view and study publicly available kernels, they are not able to directly access the intermediate artifacts, such as any preprocessed dataset or machine learning model belonging to the existing kernels. As a result, users must re-execute a kernel to generate the desired artifact.

2.2 Experiment Database

Experiment databases include data and meta-data of different data analytics and machine learning experiments executed over time [11, 12, 15, 17]. They include different information about datasets, data processing pipeline components, machine learning models, execution of machine learning training algorithms, and quality of the models. Moreover, some experiment databases allow users to store some of the artifacts generated during the execution of a workload, such as raw datasets, intermediate datasets (resulting from applying data transformation operations), and machine learning models and their hyperparameters. However, due to limited storage space, experiment databases cannot store every artifact.

Experiment databases can help in designing a better future workload. For example, users can query the database to find the answer to the following questions: what type of data transformations and model training operations are executed on a dataset and what is the accuracy of the final models? As a result, users can avoid executing data transformations or model training operations that do not result in high-quality models. Moreover, experiment databases enable reproducibility and validation of results. For example, users can query information about the environment and list of operations in a specific workload. As a result, users can re-execute the workload and compare the results.

[2]https://www.kaggle.com/c/titanic

3 COLLABORATIVE ML WORKLOAD OPTIMIZATIONS

In this section, we first define the required terms which refer to different entities in collaborative data science platforms. Then, we discuss the details of the experiment graph construction and workflow of our system and optimization process. Finally, we show how a real collaborative data science platform can utilize our system to improve the execution of machine learning workloads.

3.1 Preliminaries and Definitions

ML Task. In order to promote a collaborative effort, collaborative data science platforms must define the task which the users should solve. A task specifies the requirements and the goal of machine learning solutions. In our system, the definition of an ML task contains the following. (1) The type of the machine learning model, i.e., classification, regression, or clustering. (2) One or multiple datasets grouped into training and test (except for unsupervised tasks, which do not require a test dataset). (3) An evaluation function which assigns a score to the user-provided solution. An example of a task is to train a classification model on the training dataset D_1 which maximizes the F1 score on the evaluation test dataset D_2.

ML Workload Data and Operations. In our system, we support three types of data. (1) A *Dataset* which has one or more columns of data and is analogous to dataframe objects (e.g., pandas dataframe [10]). (2) An *Aggregate* which contains a single value or list of values. (3) A *Model* which represents a trained machine learning model.

Each data type results from one of the following three types of operations. (1) Data preprocessing and feature engineering operations which include simple data transformation and aggregation, feature selection, and feature extraction operations. This group of operations either return another Dataset (e.g., resulting from map, filter, or one-hot encoding) or an Aggregate (e.g., resulting from reduce or column correlation). (2) Model training operations which train a machine learning model on a Dataset. The result of model training

operations can either be used in other data and feature engineering operations (e.g., a PCA model) or can be used to perform prediction on a test dataset. Model training operations return a Model type. (3) Hyperparameter tuning operations which have the task of finding the best hyperparameters for a machine learning model. The most common tuning approaches are grid search, random search, and Bayesian hyperparameter search [2, 13]. In all three approaches, several models with different hyperparameters are trained and the model which performs the best is returned as the final result. In our system, instead of only returning the best performing model, we also capture all the runs with different hyperparameters. As a result, hyperparameter tuning operations returns a set of Models.

ML Workload Type. Collaborative data science platforms typically provide two modes of executions for solving tasks: *long-running workloads* and *interactive workloads*. The user can write an end-to-end script which performs data loading, data cleaning, and model training. We refer to this type of workloads as long-running workloads. Depending on the size of the initial data, a long-running workload may take anything from seconds up to many hours. The other type of workloads is more interactive. Typically, through Jupyter notebooks, collaborative data science platforms allow users to write their analysis in an interactive fashion, whereupon execution of each operation (or group of operations), they can view the result and fine-tune their analysis code. We refer to this type of workloads as interactive workloads.

3.2 Experiment Graph Representation

Workload DAG. A machine learning workload can be represented using a directed acyclic graph (DAG). In the DAG, vertices represent the artifacts, i.e., raw or preprocessed data (represented by data frame objects) and machine learning models resulting from feature engineering and model training operations and edges represent the operations in the workload. Each workload DAG has one or more initial vertices representing the raw datasets which are defined as part of the task definition in a collaborative platform. We refer to the initial vertices as the roots.

Experiment Graph. For every task, the collection of all the DAGs of the previously executed machine learning workloads forms a rooted graph (with potentially multiple root vertices) which we refer to as the *experiment graph*. More formally, we represent the experiment graph by $G(V, E)$. $V = \{v_i\}, i = 1, \cdots, n$ is the set of all the artifacts in all the workload DAGs. $E = \{e_i\}, i = 1, \cdots, m$ is the set of all the executed operations in the workload DAGs. A directed edge e from v_i to v_j in $G(V, E)$ indicates that the artifact v_j is derived from the artifact v_i by applying the operation in e. Every vertex v has the attributes $\langle f, s \rangle$ (accessed by $v.f$ and $v.s$) which represent the frequency, i.e., number of different workloads an artifact appeared in, and storage size of the artifact. Every edge e has the attribute $\langle t \rangle$ (accessed by $e.t$) which represents the run-time (in seconds) of the operation.

Inside each vertex, we store the meta-data of the artifact. Depending on how *useful* an artifact is, we may also store the actual underlying data inside the artifact (Section 4). If the artifact is a raw or a preprocessed dataset, then its meta-data includes the name, type, and total size of each column of the data and its underlying data is represented by the dataframe object (i.e., pandas dataframe [10]). If the artifact is a machine learning model, its meta-data includes the name, type, hyperparameters, and the error metric of the model and its underlying data is consist of the model weights. Each edge contains the meta-data of the operation it represents, such as the function name, training algorithm, hyperparameters, and in some cases even the source code of the operation. To uniquely identify an edge, we utilize a hash function which receives as input the operation and its hyperparameters (if it has any). Since the experiment graph is rooted, we assign a hash value to every vertex which is computed in the following way:

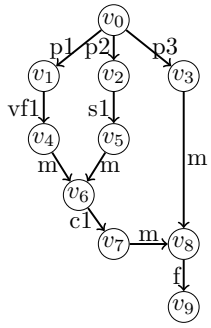
$$h(v) = \begin{cases} id, & \text{if } v \text{ is root} \\ h\Big(\sum_{e \in in\_edge(v)} (h(e.source) + h(e)) \Big), & \text{otherwise.} \end{cases}$$

where $in\_edge(v)$ returns the edges with destination v. Intuitively, the hash of a root vertex is its unique identifier (location on disk or download URL) and the hashes of other vertices are derived recursively by combining the hashes of their parents and edges which connect them to their parents.

After a machine learning workload is executed, we update the experiment graph by adding the new artifacts and operations. If any of the artifacts already exist in the graph, their frequency is updated.

Workload DAG generation. Instead of designing a new DSL, we extend the existing pandas [10] and scikit-learn [4] python packages which are frequently used for data analysis and machine learning workloads. Listing 1 shows an example of a workload script. With only a slight modification of the import commands, we are able to load our system's modules. A parser component reads the user code and instead of executing it line-by-line, it creates the edges and vertices of the workload DAG. The actual execution is invoked with *.get()* command of a vertex (Line 18, Listing 1. Figure 3a shows an example graph constructed from the code in Listing Listing 1. Table 3b shows both the label of every edge operation, i.e., time, and the hash of the operations and their hyperparameters.

```
1 import custom_pandas as pd
2
3 from custom_sklearn import svm
4 from custom_sklearn.feature_selection import SelectKBest
```



(a)

operation	label	hash
project(ad..)	$\langle 2s \rangle$	p1
project(ts, ..)	$\langle 6s \rangle$	p2
project(y)	$\langle 2s \rangle$	p3
vectorizer.f_t	$\langle 40s \rangle$	vf1
selector.f_t	$\langle 60s \rangle$	s1
concat	$\langle 10s \rangle$	c1
merge	$\langle 0s \rangle$	m
svm.fit	$\langle 100s \rangle$	f

(b)

Figure 3: Experiment graph constructed from the Listing 1 (a) and the hash of the operations in the scripts (b)

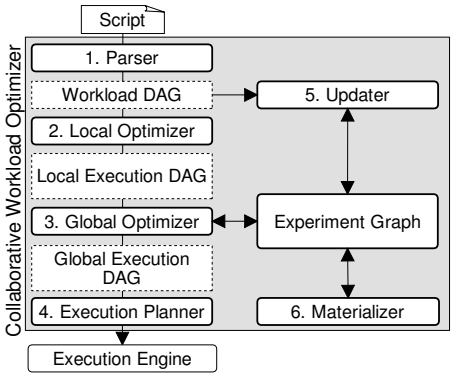


Figure 4: System overview of the collaborative workload optimizer

```
5  from custom_sklearn.feature_extraction.text import CountVectorizer
6
7  train = pd.read_csv('../input/train.csv')
8  print train.columns # [ad_desc,ts,u_id,price,y]
9  vectorizer = CountVectorizer()
10 count_vectorized = vectorizer.fit_transform(train['ad_desc'])
11 selector = SelectKBest(k=2)
12 top_features = selector.fit_transform(
13                              train[['ts','u_id','price']],
14                              train['y'])
15 top_features # print the content of the data frame
16 X = pd.concat([count_vectorized,top_features], axis = 1)
17 model = svm.SVC().fit(X, train['y'])
18 model.get()
```

Listing 1: Example script

Since at the time of the execution of the script, the experiment graph is empty, all the artifacts (vertices) have a frequency of 1. In order to represent operations which process multiple input artifacts, e.g., concat and svm.fit operations in Listing 1, we proceed as follows. First, we merge the vertices representing the artifacts into a single vertex using a merge operator. The merge operator is a logical operator which does not incur a cost, i.e., it has a run-time of 0 seconds. The merged vertex is also a logical vertex with no actual attributes which only contains the vertex ids of the merged vertices. Then, we draw an edge from the merged vertex which represents the actual operation. For example, in Figure 3a, before applying the concatenation operation, we merge v_4 and v_5 into v_6, then we apply the concatenation operation (c1). Furthermore, when computing the hash of a merged vertex, we take the merge order into account. For example, the operation svm.fit has X (represented by v_7) as first argument and train['y'] (represented by v_3) as its second argument. When computing hash of v_8, we combine the parents in the same order, i.e., $h(v_8) = h(h(v_7) + m + h(v_3) + m)$. After the DAG is constructed, its execution is invoked with the call to the *get*() command on Line 18.

3.3 System Architecture and Workflow

Figure 4 shows the components of the collaborative workload optimizer system. First, a parser component generates the workload DAG from the user scripts (Step 1). Upon the invocation of the *get*() method of an artifact, a local optimization process beings. The local optimizer extracts the subgraph which must be executed in order to compute the terminal vertex. The Local optimizer traverses the graph in reverse order starting at the terminal vertex until the root vertices. It stops the traversal when it reaches a previously computed vertex. In interactive workloads, it is likely that many of the intermediate vertices between the terminal vertex and the root vertices are previously computed. The subgraph of all the visited vertices and the edges connecting them is another DAG, which we refer to as the *local execution DAG*, and is the result of the local optimizer (Step 2). The global optimizer component receives the local execution DAG and looks for optimization opportunities, i.e., reusing materialized vertices or warmstarting model training, in the experiment graph. The result of the global optimization process is another subgraph, which we refer to as the *global execution DAG* (Step 3). Then, an execution planner receives the global execution DAG and generates the execution schedule by sorting the edges based on their topological order, which is then executed by the execution engine (Step 4). If the experiment graph is empty, then the execution planner users local execution DAG to generate the execution schedule. After the execution, an updater component updates the experiment graph to include the vertices and edges of the workload DAG (Step 5). Lastly, a materializer component decides what vertices to materialize, i.e., store their underlying data in the vertex of the graph (Step 6).

3.4 Workload Optimizer for Kaggle Use Case

In this section, we describe how our system can be integrated into Kaggle's collaborative data science platform to improve the execution of the kernels. We select the competition *Home Credit Default Risk*[3]. The task of the competition is to train a classification model which predicts whether a client is able to repay their loans. The task has 8 training datasets and 1 test datasets. The goal of the submitted solutions is to maximize the evaluation function, the area under the ROC curve between the predicted values and observed target values. After a kernel is submitted, it goes through every step of the workload optimizer system in Figure 4. For every competition, we maintain a separate experiment graph. At the time of the first kernel submission, the experiment graph is empty, therefore, the workload optimizer system skips Step 3 of Figure 4 and generates the execution plan directly from the workload execution subgraph. If the experiment graph is not empty, the global optimizer tries to optimize the workload using the experiment graph. For example, the most popular kernel[4] in the competition has been copied by more than 5000 different users. This indicates the kernel has been executed at least 5000 times, although it is very likely that many of the users run the script more than once. Each execution of the kernel takes nearly 200 seconds. In our experiments, we show that we are able to reduce the execution time to less than 10 seconds when the same kernel is executed more than once.

4 ARTIFACT MATERIALIZATION

Depending on the number of the executed workloads, the generated artifacts may require large amount of storage space. For example, in the Home Credit Default Risk Kaggle competition[5], a popular script which analyzes a dataset of 160 MB, generates up to 17 GB of artifacts. Therefore, materializing every artifact under a limited storage budget is not feasible. In this section, we introduce two algorithms for materializing the artifacts of the experiment graph under limited storage. The first algorithm utilizes both general metrics, such as the size and access frequency of vertices and execution time of the edges, and a machine learning specific metric, i.e., the quality of the terminal models, to decide what artifacts to materialize. The second algorithm is an extension of the first algorithm which also considers how the artifacts are stored on the file system. Since many of the existing operations in the experiment graph are operating on one or a small group of columns inside a data frame, the resulting artifacts have many duplicated columns. We implement a compression strategy which avoids storing duplicated columns. The second algorithm takes the duplication information into account when making the decision on what artifacts to materialize. Throughout the rest of the paper, we use the terms artifact and vertex interchangeably.

[3]https://www.kaggle.com/c/home-credit-default-risk/
[4]https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction
[5]https://www.kaggle.com/c/home-credit-default-risk

4.1 Materialization Problem Formulation

Bhattacherjee et al. [3] proposes an algorithm for efficient storage of different versions of a dataset (i.e., artifacts) under limited storage. The goal of their algorithm is to materialize the artifacts which result in the lowest recreation cost while ensuring the total size of the materialized artifacts does not exceed the storage capacity. However, there are several reasons which render their solution inapplicable to the artifact materialization problem. First, their approach only considers access frequency and reconstruction cost of an artifact. For the experiment graph, we must also consider the effect of the materialized artifacts on the efficiency of machine learning workloads, i.e., materialize artifacts which result in high-quality machine learning models. Second, their solution does not consider merge operations, e.g., join, concatenation, and model training, which are common in machine learning workloads. Lastly, their solution considers a scenario where new artifacts are rarely added. In the machine learning workload optimization, new artifacts are continuously added to the experiment graph. As a result, a proper solution must accommodate the addition of new artifacts.

Here, we first formulate the problem of artifact materialization as a multi-objective optimization problem, with the goal of minimizing two functions which consider both the artifact recreation cost and the model quality of the artifacts, given the storage requirement constraint.

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

$$WC(G) \coloneqq \sum_{v \in V} (1 - v.mat) \times v.f \times in\_edge(v).t$$

where $v.mat = 1$ if artifact v is materialized and 0 otherwise, $v.f$ is the frequency of the artifact v, and $in\_edge(v).t$ returns the edges with destination v and run-time t. Merged vertices have no impact on $WC(G)$ since they carry no actual data content except for pointers to the original vertices, therefore, we are not computing the weighted recreation cost for the merged vertices. Intuitively, the weighted recreation cost computes the total amount of execution time required to recompute all the vertices while considering their frequencies. Materialized artifacts incur no cost since they are stored. Non-materialized artifacts incur a cost equal to the execution time of the proceeding operations multiplied by their frequency. For example, in Figure 3, if we do not materialize v_4 and assuming it has a frequency of 2, we must

consider both executions of the operation *vectorizer.f_t* when computing the weighted cost. Whereas, if v_4 is materialized, the *vectorizer.f_t* operation has no impact on the weighted recreation cost.

Estimated Quality Function (EQ). In order to define the estimated quality function, we need to define the followings first.

$$M(G) := \{v \in V \mid v \text{ is a terminal model}\}$$

is the set of all the terminal models in the experiment graph. For every vertex v in the graph,

$$M(G, v) := \{m \in M(G) \mid (v = m) \vee (v \text{ is connected to } m)\}$$

is either v itself, when v is a terminal model, or the set of all terminal models to which v is connected.

$$p(G, v) := \begin{cases} quality(v), & \text{if } v \in M(G) \\ \max\limits_{m \in M(v)} \left(\dfrac{quality(m)}{cost(v, m)}\right), & \text{otherwise} \end{cases}$$

is the potential of an artifact, where $quality(m)$ represents the quality of a terminal model measured by the evaluation function of the task. We define $cost(v, m)$ as the total time of executing all the operations starting from vertex v until model m,

$$cost(v, m) := \sum_{e \in path(v,m)} e.t$$

If v itself is a model, then $p(G, v) = quality(v)$. When v is not a model, we first compute the ratio of quality over cost for all of v's connected models and the potential of v to the ratio with the largest value. Intuitively, a high potential artifact is an artifact which results in a high-quality terminal model with low cost.

Now, we define the estimated quality function as:

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

Multi-Objective Optimization. Given the two functions, i.e., weighted recreation cost and estimated quality, we would like to find the optimal 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):

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

Bhattacherjee et al. prove minimizing the recreation cost alone is an NP-Hard problem [3]. While there are different approximate strategies for solving multi-objective optimization problems [5], they are time-consuming, which renders them inappropriate to our setting, where new artifacts are constantly added to the graph. Execution of every workload results in an update to the experiment graph, 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 the experiment graph.

4.2 ML-Based Greedy Algorithm

We propose a greedy heuristic-based algorithm for materializing the artifacts in the experiment graph which aims to minimize the weighted recreation cost function and maximize the estimated quality function. Every task T is associated with an experiment graph. Each task also has a storage budget and runs a separate instance of the materialization algorithm. For example, in Figure **??**, there are three separate experiment graphs for the competitions A, B, and C, each having a dedicated storage budget and running a separate materializer component.

Algorithm 1: Artifacts-Materialization

Input: G_E: experiment graph, $\mathcal{B}$: storage budget
Output: experiment graph with materialized vertices

```
1  S := 0;                          // size of the materialized artifacts
2  for v ← roots(G_E) do
3      if v.mat = 0 then
4          v.mat := 1;
5          S := S + v.s;
6  PQ := empty priority queue;
7  for v ← V do
8      if v.mat = 0 then
9          utility := U(G_E, v);
10         PQ.insert(v);            // sorted by utility
11 while PQ.not_empty() do
12     v := PQ.pop();               // vertex with max utility
13     if S + v.s ≤ B then
14         v.mat := 1;
15         S := S + v.s;
```

Algorithm 1 shows the details of our method for selecting the vertices to materialize. First, we start by materializing all the root vertices. This is essential as many of the feature engineering and model building operations are not invertible. As a result, we cannot reconstruct the raw datasets if they are not materialized. Then, for every non-materialized vertex, we compute the utility value of the vertex (Lines 7-10). Then, we start materializing all the vertices, sorted by their utility, until the storage budget is exhausted (Lines 11-15). The utility function ($\mathcal{U}(G, v)$) computes the goodness of an artifact with

respect to its recreation cost, how often it is reused, and the estimated quality gained from the artifact. To define the utility function, we first define the recreation cost function as the following:

$$rc(G, v) \coloneqq \sum_{e \in \bigcup_{v_0 \in roots} path(v_0, v)} e.t$$

which is the total execution time of all the operations from the root vertices to v. We also define $trf(v)$ as the amount of time required to transfer vertex v from the experiment graph to the machine running the workload. The trf function depends on the size of the vertex and the infrastructure type of the system. Then, the utility function is defined as:

$$\mathcal{U}(G, v) \coloneqq \begin{cases} 0, & \text{if } trf(v) \geq rc(G, v) \\ \dfrac{v.f \times p(G, v) \times rc(G, v)}{v.s}, & \text{otherwise.} \end{cases}$$

where $p(G, v)$ and $rc(G, v)$ are the potential and recreation cost of v and $v.f$ and $v.s$ are the frequency and size of the vertex. If transfer cost is greater than the recomputation cost, we do not materialize the vertex since re-executing the operations is faster than copying the vertex. Taking the transfer 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., in-memory or on-disk experiment graphs). If recreation cost dominates the transfer time, then, we materialize vertices which are more costly to recompute, have larger impacts on the overall quality of the experiment graph, and have a higher frequency.

Run-time and Complexity. The complexity of the materialization algorithm is $O(|V|)$ where $|V|$ is the number of vertices in the experiment graph. As users execute more workloads, the size of the experiment graph increases and running the materialization algorithm becomes more costly. However, once the utility of a vertex is computed, it does not change until it appears in a new user workload. Therefore, in our implementation, we compute the utilities for all the new vertices (Lines 7 - 10, Algorithm 1) once and only recompute the utility if the vertex appears in a new workload. By precomputing the utilities we reduce the complexity of each run of the materialization algorithm to $O(|W|)$, where $|W|$ is the number of vertices in the workload DAG.

4.3 Storage-Aware Materialization Algorithm

Since many feature engineering operations only operate on one or a few columns of a dataset, the resulting artifact of a feature engineering may contain many of the columns of the input artifact. As a result, after materialization, there are many duplicated columns across different artifacts. To further reduce the storage cost, we implement a deduplication mechanism. We assign a unique hash to every columns of the artifacts. When executing an operation on an artifact, all the columns of the resulting artifact, except for the ones affected by the operation carry the same hash value. When storing an artifact, the storage manager unit examines the hash of every column, and only stores the columns that do not exist in the storage unit. The storage manager tracks the column hashes of all the artifacts in the experiment graph. When a specific artifact is requested, the storage manager combines all the columns which belong to the artifact into a data frame and returns the data frame. This results in a large decrease in the storage cost (e.g., for the same script of the Home Credit Default Risk Kaggle competition[6] which generates 17 GB of artifacts, deduplication result in only 8 GB of storage).

Greedy Meta-Algorithm. We propose a storage aware materialization meta-algorithm (Algorithm 2) which iteratively invokes Algorithm 1 (Artifact-Materialization). We define a variable to represent the remaining budget (Line 1). While the budget is not exhausted, we proceed as follows. We extract the current set of materialized nodes from the graph (Line 3), then we apply the Artifact-Materialization algorithm using the remaining budget to compute new vertices for materialization. If the Artifact-Materialization algorithm did not find any new vertices to materialize, we return the current graph (Line 6). We compute the compressed size of the graph artifacts (Line 7), which computes the size of graph artifacts after deduplication. Next, we update the required storage size of the remaining artifacts (Line 8). For example, if the materialized artifact v_1 contains some of the columns of the non-materialized artifact v_2, then we only need to store the remaining columns of v_2 to fully materialize it. Therefore, we update the size of v_2 to indicate the amount of storage it requires to fully materialize. Finally, we compute the remaining budget by deducting the compressed size from the initial budget.

5 REUSE AND WARMSTARTING OPTIMIZATIONS

With the experiment graph constructed and materialized, our collaborative optimizer looks for opportunities to reuse existing materialized artifacts in the experiment graph and warmstart model training operations. Figure 5 shows an example of the optimization process for reuse and warmstarting. In the workload DAG, the user invokes the *.get*() command on vertex 7, which represents a machine learning model. To compute the local execution DAG, the local optimizer scans the workload DAG to find previously computed

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

Algorithm 2: Storage-aware Materialization

Input: G_E: experiment graph, $\mathcal{B}$: storage budget
Output: experiment graph with materialized vertices

```
1 R := B ;
2 while R > 0 do
3     prev_mats := materialized_nodes(G_E);
4     G_E := Artifact-Materialization(G_E, R);
5     if prev_mats = materialized_nodes(G_E) then
6         return G_E;
7     compressed_size := deduplicate(G_E);
8     update_required_size(G_E);
9     R := B − compressed_size;
```

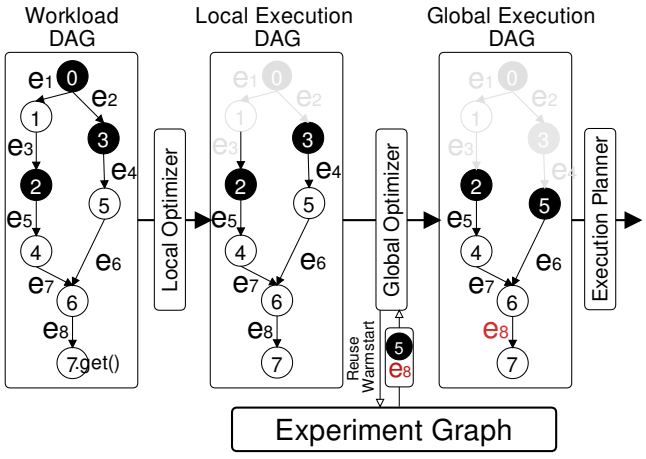


Figure 5: Reuse and Warmstarting Optimizations

vertices. This is an important step for interactive workloads. In this example, besides from the root vertex (vertex 0), vertex 2 and 3 are already computed (represented by black color). Therefore, the local optimizer prunes vertex 0 and edges e_1 and e_2. Then, the global optimizer looks for reuse and warmstarting opportunities in the experiment graph. In this example, vertex 5 was computed in a previous workload and its result was materialized in the experiment graph. Similarly, the model training operation e_8 was already executed in an existing workload. The global optimizer transfers vertex 5 to the workload, which results in further pruning of vertex 3 and edge e_4. The global optimizer also warmstarts e_8 which increases the convergence rate of the training operation. Both Reuse and Warmstarting can reduce the total execution time of the workload. In the rest of this section, we present the details of both Reuse and Warmstarting methods.

5.1 Reuse

Since during the materialization process, we ensure the recomputation cost of a materialized vertex outweighs its transfer cost (i.e. $trf(v) \geq rc(G, v)$), we can safely transfer the materialized vertex to the machine executing the workload and guarantee the total execution time will improve.

The global optimizer component compares the local execution DAG with the experiment graph to find any vertices that are materialized in the experiment graph and returns them instead of computing them. The vertex hashing procedure in Section 3 indicates that two vertices from two different graphs are the same if they have the same id (hash value). Therefore, to find matching vertices in the local execution DAG, global optimizer queries their ids in the experiment graph.

However, in cases where querying the experiment graph incurs a penalty, such as when the experiment graph is stored on a remote computing machine or there are many concurrent workloads being executed in the collaborative platform, we must minimize the number of queries to the experiment graph. In this section, we present three approaches for searching the experiment graph for materialized vertices, namely, BottomUp reuse, TopDown reuse, and Hybrid reuse.

BottomUp reuse. The process of BottomUp reuse is similar to how we construct the local execution DAG. Algorithm 3 shows the details of the BottomUp reuse. The algorithm receives the local execution DAG ($G_L(V_L, E_L)$), the terminal vertex v_t whose data is requested, and the experiment graph ($G_E(V_E, E_E)$). The BottomUp reuse utilizes the following early-stopping principle. If a vertex from in G_L exist in G_E and is materialized ($is\_mat(G_E, v)$ function returns true if v is materialized in G_E), then we skip traversing its parents and add the vertex to the set of materialized vertices. Therefore, in Line 1, if v_t is materialized, we return it as the result and in Lines 8-11, we only continue the search if the vertex (v) is not materialized. For any other vertex which is not materialized, we recursively examine its parents until there are no more vertices left (i.e., we reach the root vertices).

BottomUp reuse performs well when the terminal vertex or vertices close to the terminal are materialized in the experiment graph. However, in extreme cases, where none of the vertices of the local execution DAG are in the experiment graph, BottomUp reuse still has to examine all the vertices. Therefore, BottomUp reuse has a complexity of $O(|V_L|)$, where $|V_L|$ is the number of vertices in the local execution DAG, which means the global optimizer may make up to $|V_L|$ calls to the experiment graph.

TopDown reuse. Contrary to the BottomUp reuse, in TopDown, we start traversing the local execution DAG, from its root vertices.

Algorithm 3: BottomUp Reuse

Input: v_t: terminal vertex, G_L: local execution DAG, G_E: experiment graph
Output: set of materialized vertices $\mathcal{M}$ for reuse

```
1  if is_mat(G_E, v_t) then
2      return {v_t};
3  Q := Queue(v_t);
4  M := ∅;
5  while Q.not_empty() do
6      cur := Q.pop();
7      for v ∈ parents(G_L, cur) do
8          if is_mat(G_E, v) then
9              M.append(v);
10         else
11             Q.add(v);
12 return M;
```

Algorithm 4: TopDown Reuse

Input: v_t: terminal vertex, G_L: local execution DAG, G_E: experiment graph
Output: set of materialized vertices $\mathcal{M}$ for reuse

```
1  R = roots(G_L);
2  M := ∅;
3  for r ∈ R do
4      if r ∉ G_E then
5          continue;                    // skip this root
6      Q := Queue(r);
7      if is_mat(G_E, r) then
8          M.append(r);
9      while Q.not_empty() do
10         cur := Q.pop();
11         for v ∈ children(G_L, cur) do
12             if is_mat(G_E, v) then
13                 M.append(v) ;
14             if v ∈ G_E then
15                 Q.add(v);
16 return M;
```

Algorithm 4 shows the details of the TopDown reuse algorithm. The TopDown reuse operates on the following early-stopping principle. If a vertex from G_L does not exist in the experiment graph (G_E), we skip the traversal of its children. This follows from the graph construction procedure, where each vertex is derived from its parents, as a result, it is impossible for a child vertex to exist in a graph where its parents do not. Therefore, in Lines 4 and 14, we stop the traversal if the vertex is not in G_E. Since a workload execution graph may have multiple root vertices, the TopDown reuse algorithm first finds all the root vertices (Line 1). Then, for every root vertex, TopDown examines all of its children and add them to the set of materialized vertices if they are in the experiment graph and are materialized (*is_mat* returns true). Unlike the BottomUp approach, when a node is materialized, we cannot stop the traversal, since a materialized vertex may also have materialized children.

TopDown performs well in scenarios where the vertices close to the root are materialized. If the current workload operates on a completely new root (which never appeared in the experiment graph) or the workload contains early data exploration which never appeared in experiment graph, TopDown reuse will quickly stop the search process. However, in extreme cases, where the terminal vertex is materialized in the experiment graph (i.e., a workload is re-executed), then TopDown must traverse the entire local execution DAG. Therefore, TopDown reuse also has a complexity of $O(|V_L|)$ and makes at most $|V_L|$ calls to the experiment graph.

Hybrid reuse still needs a bit of work, I'm currently implementing it to see its impact and if it is even useful

Hybrid reuse. Both BottomUp and TopDown reuse perform well in specific scenarios. However, neither of them can adapt to the different characteristics of a workload (e.g., how similar a new workload is to the previous workloads or how large the execution DAG is). We devise a dynamic reuse approach, called Hybrid reuse, which adapts to the current workload. Algorithm 5 shows the process of Hybrid reuse. Hybrid reuse combines the two early-stopping principles utilized in TopDown and BottomUp reuse algorithms to prune as many vertices without querying the experiment graph. The two methods $R\_BFS(v, G, n)$ and $F\_BFS(v, G, n)$, perform a breadth-first-search traversal starting from vertex v and return the vertex after n visits. *R_BFS* traverses in the reverse direction of the edges (visiting the parents of the vertex) and *F_BFS* traverses in along the direction of the edges (visiting children of the vertex). It is important to emphasize that the two methods, *R_BFS* and *F_BFS*, do not make calls to the experiment graph and only return a vertex in the local execution DAG after the specified visits.

First, Hyrbid reuse traverses G_L starting from the terminal node in reverse (*R_BFS*, Line 4) until it visits half of the vertices ($|V_L|/2$). Hybrid reuse then iteratively prunes half of the remaining vertices and adds any materialized vertex from the experiment graph to the set of materialized vertices. In Line 7, if the vertex is not in the experiment graph, using the TopDown early-stopping principle, the algorithm prunes the bottom half of the graph and search in the top half (i.e., *R_BFS* on Line 8). If the vertex is materialized (Line 16), the

algorithm first adds it to the list of materialized vertices, then uses the BottomUp early-stopping principle to prune the top half the graph (i.e., *F_BFS* on Line 18). When the vertex is in the experiment graph but it is not materialized (Line 9), the algorithm cannot safely prune the graph, as materialized vertices can still appear in either top half or bottom half, or both. Hybrid reuse utilizes the following heuristic to decide which half of the graph to prune. First, it computes the average value of the utility function for the parents and children of the vertex (Lines 10 and 11). If the average utility of the parents is larger, then it prunes the bottom half, otherwise, it prunes the top half. The intuition behind this heuristic is the following. The materialization algorithm always materializes vertices based on their utility value. Therefore, if the utility of parents of a vertex is larger than the utility of its children, then there is a higher likelihood that more vertices in the top half of the graph are materialized.

Algorithm 5: Hybrid Reuse

Input: v_t: terminal vertex, G_L: local execution DAG, G_E: experiment graph
Output: set of materialized vertices $\mathcal{M}$ for reuse

```
1  N := |V_L|;
2  step := 2;
3  M := ∅;
4  v := R_BFS(v_t, G_L, N/step);
5  while step ≤ log(N) do
6      step = step × 2;
7      if v ∉ G_E then
8          v := R_BFS(v, G_L, N/step);
9      else if v ∈ G_E ∧ ¬is_mat(G_E, v) then
10         prev := avg(U(parents(G_E v));
11         next := avg(U(children(G_E, v));
12         if prev ≥ next then
13             v := R_BFS(v, G_L, N/step);
14         else
15             v := F_BFS(v, G_L, N/step);
16     else if is_math(G_E, v) then
17         M.append(v);
18         v := F_BFS(v, G_L, N/step);
19 return M;
```

The advantage of the Hybrid reuse is that it requires at most $log(|V_L|)$ calls to the experiment graph since in every iteration we are pruning half of the graph. This is in contrast to the TopDown and BottomUp reuses, wherein certain scenarios they may require $|V_L|$ calls to the experiment graph.

5.2 Warmstarting

Many model training operations include random processes. For example, in random forests, to decide when to split a tree node, features are randomly permuted. A random seed parameter controls the random behavior. Two training operations on the same dataset with the same hyperparameters may result in completely different models if the random seeds are different. Therefore, the Reuse algorithm is not able to find the previously trained model, if the random seeds are different. Instead, we try to warmstart model training operations using the existing models in the experiment graph. 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 [1].

During the graph construction, for every model training operation, we compute an extra hash value, which does not consider the random seed parameter. We refer to this hash value as the seedless hash. The seedless hash allows us to find similar training operations that only differ in the random seed. When the Reuse algorithm encounters a machine learning model vertex, two scenarios can occur. In the first scenario, a similar model vertex with the same id also exists in the experiment graph. In this scenario, we can safely reuse the model vertex since the only way for both model vertices to have the same id is that both models are trained on the same data using the same training operations with the equal hyperparameters and random seeds. In the second scenario, the machine learning model vertex does not exist in the experiment graph. In this scenario, we utilize Algorithm 6 to detect whether warmstarting is possible. The warmstarting algorithm receives the model vertex v_m, the local execution DAG, and the experiment graph as inputs and tries to warmstart the training operation for v_m with a model from the experiment graph. The algorithm first finds the parent of the model vertex, represented by $v_{dataset}$ on Line 1 and the training operation, represented by e_{train} on Line 2. $v_{dataset}$ is the dataset used in the operation e_{train}. To warmstart e_{train}, the algorithm first ensures $v_{dataset}$ is in the experiment graph (Line 4). Then, for every outgoing edge of the $v_{dataset}$, the algorithm compares the seedless hash of the edge with the seedless hash of e_{train}. In the algorithm, the function *sl_hash* computes the seedless hash of an edge. Equal seedless hashes indicate that the training operation from the experiment graph only differs in the random seed when compared to e_{train}. Therefore, the result of the training operation in the experiment graph is a candidate for warmstarting e_{train} (Lines 6-8). In case there are more than one wamrstarting candidates, we select the candidate model with the maximum quality to warmstart the training operation (Lines 9-11).

Algorithm 6: Warmstarting

Input: v_m: model vertex, G_L: local execution DAG, G_E: experiment graph
Output: modified G_L with warmstarted training

```
v_dataset := parent(G_L, v_m);
e_train := edge(G_L, v_dataset, v_m);
C := ∅;                                  // set of candidate models
if v_dataset ∈ G_E then
    for e ∈ out_edges(G_E, v_dataset) do
        if sl_hash(e) = sl_hash(e_train) then
            m := e.dest;
            C.add(m);
if C.not_empty() then
    m := argmax_{m∈C} quality(m);
    warmstart(e_train, m);
```

6 EVALUATION

INCOMPLETE, No need to read!!

In this section, we evaluate the performance of our proposed optimizations.

6.1 Setup

To evaluate our proposed optimizations, we provide two different set of workloads, namely, OpenML workload and Kaggle workload.

OpenML workloads: In the OpenML workloads, we utilize some of the popular machine learning pipelines from the OpenML repository for solving task 31, i.e., classifying customers as good or bad credit risks using the German Credit data from the UCI repository [6][7]. Table 1 shows the id, components, and number of executions of the OpenML pipelines.[8]

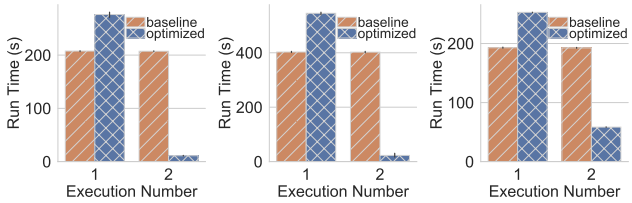
Kaggle workloads: In the second set of workloads, we target interactive machine learning loads where a large portion of the analysis is spent on data and feature preprocessing. We use some of the popular scripts in the **Home Credit Default Risk** competition in Kaggle. The competition comprises of 9 datasets with a total size of 2.7 GB. The goal of the competition is to train a classification model which predicts whether clients are able to repay their loans. We utilize scripts written by Kaggle users and design our own scripts to measure the improvement for reuse, warmstarting, and hyperparameter tuning when using the experiment graph.

[7]https://www.openml.org/t/31

[8]information about each pipeline is available at https://www.openml.org/f/id

id	operations	#exec
5981	Imputer→ Standard scaler→Logistic regression	11
7707	Imputer→Onehot encoder→Standard scaler →Variance thresholder→SVM	594
8315	Imputer→Onehot encoder →Variance thresholder→Random Forest	1084
8353	Imputer→Onehot encoder →Variance thresholder →Svm	1000
8568	Imputer→Onehot encoder →Variance thresholder→Random Forest	555

Table 1: OpenML pipeline descriptions.



(a) Feature1 **(b) Feature2** **(c) IntroFeature**

Figure 6: Run time of repeated executions of the Kaggle scripts.

6.2 Reuse and Warmstarting

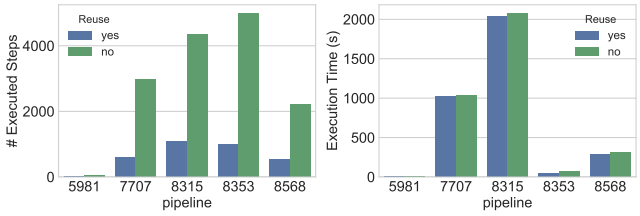
6.2.1 Kaggle Workloads. To evaluate the performance of reuse and warmstarting optimizations, we selected three popular scripts from the Home Credit Kaggle competition. Script Feature1[9], Feature2[10], and Script IntroFeature[11] read the input data, perform several steps of preprocessing and feature extractions with many visualization steps and train multiple classification models to use the preprocessed and generated features. The scripts have been copied by other users more than 6,500 times. In the first experiment, we measure the run time of repeated executions of the same script. The reported results are averaged over 5 (error bar included).

Figure 6 shows the run time of repeated runs of Scripts Feature1, Feature2, and IntroFeature. Since there experiment graph is not yet populated, the first execution of both versions of the scripts (optimized and baseline) have similar run times. However, after the first run, the experiment graph is populated with the workload graphs of each script and the reuse optimization returns the requested data without executing the operations. This reduces the run time to under

[9]https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering

[10]https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering-p2

[11]https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction



(a) Executed Steps (b) Execution Time

Figure 7: Effect of the reuse optimization on the total number of executed transformations and the total execution time for every OpenML pipeline

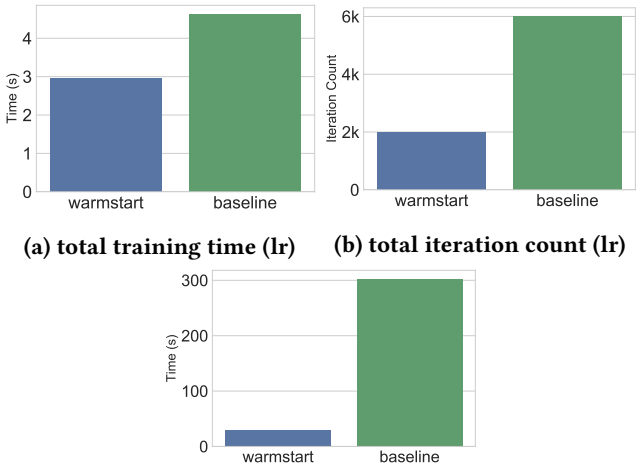
10 seconds for Scripts Feature1 and Feature2 and 60 seconds for Script IntroFeature. Most of the run time is spent on returning the requested data and visualization. However, the baseline approach (no experiment graph) does not gain performance improvement from previous runs.

6.2.2 Feature Processing Reuse. In Figure 7, we show the effect of the reuse optimization on the OpenML workflow. The number of executed steps drastically decreases as the majority of the pipelines have the exact same data transformation steps and they only differ in the hyperparameters of the model (Figure 7a). However, Figure 7b shows that the reuse optimization does not impact the total execution time. This is specific to the OpenML use case, as the model training time dominates the data transformation time.

The figure only shows estimates

6.2.3 Model Warmstarting. In this experiment, we study the effect of the model warmstarting optimization on two pipelines (pipelines 5891 and 8568) from the openml database. Pipeline 5891 has a logistic regression model. There a total of 11 configurations in the database. The stopping condition for the logistic regression model is the convergence tolerance. Pipelines 8568 has random forest model. There are a total 555 configurations for pipelines 8568. The training of the random forest stops when the number of samples in any leaf node is below a user-defined threshold.

Figure 8 shows the result of the model warmstarting optimization on two types of models in the experiment database. Figures 8a and 8b shows the effect of warmstarting on the logistic regression model. Since the data size is small, the training time is fast and the total time is mostly dominated by the data processing and start-up time. To better show the effect of the warmstarting on the model training, we also include the total number of iterations for training the model on all 11 configurations. The warmstarting optimization reduces the number of iterations by a factor of three. Figure 8c shows the total training time for all the 555 random forest models. The warmstarting optimization reduces the total



(a) total training time (lr) (b) total iteration count (lr)

(c) total training time (rf)

Figure 8: Effect of the warmstarting optimization on the total training time and iteration count. In (a) and (b) we train 11 logistic regression models and in (c) we train 555 random forest model from the configurations that exist in the experiment data.

training time by one order of magnitude (from 300 seconds to 30 seconds).

6.2.4 Combined Optimization. In this section we study the effect of combining both optimizations (Reuse and Warmstarting).

6.3 Evaluation of the Improved Hyperparameter Tuning

6.3.1 Search Space Proposal.

6.3.2 Improved Bayesian Hyperparameter Tuning. In this experiment, we focus on several of the popular machine learning pipelines (flow 7707, 8353, 8315) designed for solving task 31[12], classifying customers as good or bad credit risks using the German Credit data from the UCI repository [6]. We extracted the meta-data from the OpenML database which includes all the executions of the pipeline, the value of the hyperparameters, and the evaluation metrics. Using the meta-data, we initialize the hyperparameter optimization process with the values of the hyperparameters for each execution and the loss ($1 - accuracy$) for the specific execution. We then execute the search with a budget of 100 trials, trying to minimize the loss of the OpenML pipelines. We repeat this experiment 10 times, for every pipeline. Figure 9 shows the average of losses of the 100 trials for the 10 experiments.

[12]https://www.openml.org/t/31

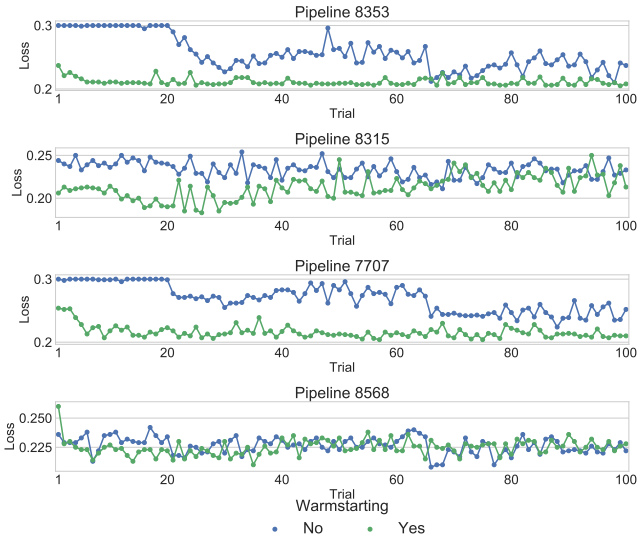


Figure 9: Loss value of 100 Trials with and without warmstarting

Warm starting the search decreases the overall loss of the trials.

7 RELATED WORK

- repository of ml experiments
- Scientific workflow systems?
- Hyperparameter tuning, random and grid search
- Dataset versioning and materialization [3, 16]
- Transfer learning

8 CONCLUSIONS

REFERENCES

[1] 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*. ACM, 1387–1395.

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

[3] 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.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 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.

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

[6] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

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

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

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 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.

[10] 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.

[11] Hui Miao and Amol Deshpande. 2018. ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows. *Data Engineering* (2018), 26.

[12] 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*.

[13] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[14] 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

[15] 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.

[16] 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*. ACM, 1285–1300.

[17] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. Model DB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM, 14.