Optimization of Machine Learning Workloads with Experiment Databases

ABSTRACT

Collaborative data science platforms, such as Google Colaboratory and Kaggle, affected the way users solve machine learning tasks. Instead of solving a task in isolation, users write their machine learning workloads and execute them on these platforms and share the workloads with others. This enables other users to learn from, modify, and make improvements to the existing workloads. However, this collaborative approach suffers from two inefficiencies. First, storing all the artifacts, such as raw datasets, generated features, and models with their hyperparameters, requires massive amounts of storage. As a result, only some of the artifacts such as scripts, operations, and models are stored and users must re-execute the scripts and operations to reconstruct the desired artifact. Second, even if all the artifacts are stored, manually finding desired artifacts is a time-consuming process.

The contributions of this paper are two-fold. First, we utilize a graph to store the artifacts and operations of machine learning workloads as vertices and edges respectively. Since storing all the artifacts is not feasible, we propose two algorithms for selecting the artifacts with high expected rates of future reuse. We then store the selected artifacts in memory for quick access, a process which we call artifact materialization. The algorithms consider several metrics, such as access frequency, size of the artifact, and quality of machine learning model artifacts to decide what artifacts to materialize. Second, using the graph, we propose three optimizations, namely reuse, model warmstarting, and fast hyperparameter tuning, to speed up the execution of the future workloads and increase the efficiency of collaborative data science platforms.

PVLDB Reference Format:
. . *PVLDB*, (): xxxx-yyyy, .
DOI:

1. INTRODUCTION

Proceedings of the VLDB Endowment, Vol. , No.
ISSN 2150-8097.
DOI:

Machine learning is the process of analyzing training datasets to extract features and build a machine learning model using these features to solve a specific task, such as automatically labeling images based on the image content. 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. Recent collaborative data science platforms facilitate workload sharing. As a result, data scientists can study other workloads, learn from them, and improve upon them to train better machine learning models. Some platforms such as Kaggle [2] and Google Colabratory [1] provide an intuitive way of sharing scripts and raw data and allow execution of the scripts on the platform using Jupyter notebooks [8]. Other platforms such as OpenML [14], ModelDB [16], and ProvDB [9] enable the data scientists to store the operations and artifacts of their machine learning workloads, i.e., raw datasets, intermediate datasets resulting from the operations, and models with their hyperparameters. These platforms are typically referred to as experiment databases [13]. Data scientists can then query other users' operations and artifacts to search for more detailed information such as the types of operations, preprocessed datasets, models, hyperparameters, and evaluation metrics for specific machine learning tasks.

While collaborative approaches lead to better performing machine learning models, they introduce two problems. First, since storing all the artifacts requires a massive storage unit, existing platforms only allow the storage of raw datasets, the operations or the scripts, final models with their hyperparameters, and a limited number of intermediate artifacts. For example, a popular script on the **Home Credit Default Risk**[1] competition in the Kaggle platform which processes a training dataset of 160 MBs in size generates up to 17 GBs of artifacts. However, one common usage pattern in the collaborative data science platforms is for users to select an existing intermediate artifact (e.g., a preprocessed dataset) and continue their own analysis and model training on the artifact. Since the intermediate artifact is not available, the user must rerun the script to generate the desired artifact.

Second, even if all the artifacts are stored, going through the available scripts or manually querying the experiment databases is time-consuming. As a result, many data scientists execute their machine learning workload without check-

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

ing if they can reuse part of the workload from the existing ones.

We propose a solution for efficiently storing the artifacts inside a graph, where vertices are the artifacts and edges are the operations connecting the artifacts and automatically optimize new workloads using the graph. We propose two algorithms to select promising artifacts and store them in memory for quick access, a process which we refer to as the artifact materialization. The first algorithm utilizes two different types of metrics for materializing the artifacts, i.e., general and machine learning specific metrics. The general metrics include the size and access frequency of the artifacts and the run-time of the operations. The machine learning specific metric consists of the quality score of the machine learning models resulting from the artifact. Since many of the artifacts have overlapping data columns, we perform column deduplication and remove duplicated columns before storing them. The second algorithm is storage-aware, i.e., it takes into account the deduplication information before deciding on what artifacts to materialize. Using the graph of the materialized artifacts, which we refer to as the *experiment graph*, we automatically extract information to optimize the process of design and execution of future machine learning workloads. Specifically, we provide three optimizations, namely, reuse, model warmstarting, and fast hyperparameter tuning. In reuse, we look for opportunities to reuse an existing materialized artifact to avoid data reprocessing. Reuse decreases the data processing time and enables interactive data analysis, especially during the initial exploratory data analysis phase where many data scientists perform similar data transformation, aggregation, and summarization operations on the data. In model warmstarting, we devise a method to detect if we can warmstart the model in the workload with an existing materialized model artifact. Warmstarting speeds up the convergence rate, resulting in shorter training time. Lastly, we propose to utilize the experiment graph to improve the process of hyperparameter tuning. There are 3 common hyperparameter tuning techniques, namely, grid search, random search, and Bayesian hyperparameter search [7, 12]. All three methods rely on defining a search space before the process of tuning begins. Setting the search space typically requires a combination of domain knowledge and machine learning expertise. As a result, non-expert users struggle with setting an effective search space. Moreover, after setting the search space, the Bayesian hyperparameter tuning technique requires many initial trials until it starts to propose promising hyperparameters. Using the existing model artifacts inside the experiment graph, we construct a feasible search space for hyperparameter tuning and initialize the process of the Bayesian hyperparameter tuning. As a result, data scientists can utilize the automatically defined search space for hyperparameter tuning (applicable to grid, random, and Bayesian search) and skip the initial trials of the Bayesian hyperparameter tuning.

In summary, we make the following contributions:

- We present a graph representation of artifacts and operations of machine learning workloads.
- We propose algorithms for materializing the workloads artifacts in the experiment graph under limited storage capacity

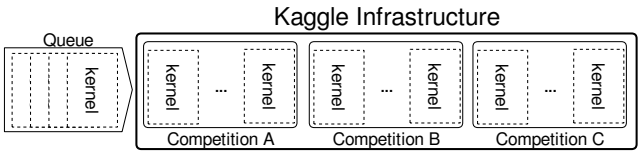


Figure 1: Kaggle Infrastructure

- We propose automatic reuse and model warmstarting, as well as fast hyperparameter tuning for new workloads using the experiment graph.

The rest of this document is organized as follows.

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 or Jupyter notebooks 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. Kaggle groups kernels by competition. Each kernel has limited CPU, GPU, disk space, and memory (i.e., 4 CPU cores, 17 GB of RAM, 5 GB of disk space, and a maximum of 9 hours of execution time. GPU kernels have 2 CPU cores and 14 GB of RAM[2]). In busy times, this results in users to be placed in queues (especially for GPU-enabled machines) until resources become available. Figure 1 shows the infrastructure of Kaggle.

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 pipeline 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 example, in the *Titanic: Machine Learning from Disaster* competition in Kaggle[3], 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

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

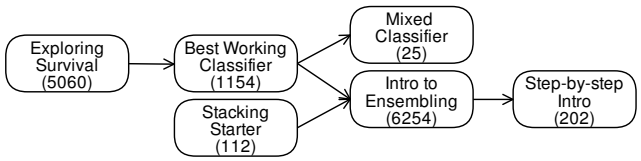


Figure 2: The fork hierarchy of some of the popular kernels in Kaggle's Titanic competition

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 [9, 14, 10, 16]. 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 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.

3. ML WORKLOAD OPTIMIZATIONS

In this section, we first describe the common types of operations in machine learning workloads. Then, we discuss how to capture and store the operations in the experiment graph. Lastly, we discuss how do we utilize the experiment graph to optimize new workloads.

3.1 Operations in ML Workloads

Feature Extraction	Feature Selection
feature hasher	variance threshold
one hot encoding	select k best
count vectorizer	select percentile
tfidf transformer	recursive feature elimination
hashing vectorizer	select from model
extract_patch_2d	

Table 1: List of feature extraction and feature selection operations

We assume the main units of work are data frame (e.g., Pandas, R Data Frames, and Spark Data Frames) like objects that contain one or many columns, where all the data items in one column are of the same data type. We divide the operations in the ML workloads into 3 categories.

3.1.1 *Data and Feature Engineering*

This group of operations typically belongs to three categories, i.e., simple data transformations and aggregations, feature selection, and feature extraction. All of these operations, receive one or multiple columns of a dataset and return another dataset as result. While different data processing tools may provide specialized data transformation and aggregation operations for data frame objects, most of them provide the same or similar operations such as map, reduce, group by, concatenation, and join. In Table 1, we show a list of the most common feature extraction and feature selection operations.

3.1.2 *Model Training*

Model training operations are a group of operations that receive a dataset (or one or multiple columns of a dataset) and return a machine learning model. The result of model training operations can either be used in other data and feature engineering operations (e.g., applying PCA to reduce the number of dimensions of the data) or can be used to perform prediction (for classification and regression tasks) on unseen data.

3.1.3 *Hyperparameter Tuning*

Before training a machine learning model, one has to set the hyperparameters of the model to appropriate values. Typically, the best values for the hyperparameters of a model vary across different datasets. The goal of hyperparameter tuning operations is to find the set of hyperparameters that yield the model with the best performance on a dataset. A hyperparameter tuning operation is defined by a budget, a search method, and a search space. The budget specifies how many models with different hyperparameter values, within the specified search space, should be trained and the search method specifies what search strategy should to incorporate. The most common search methods are grid search, random search, and Bayesian hyperparameter search [3, 12].

3.2 Experiment Graph Representation

To efficiently apply our optimizations, we utilize a graph data structure, called the experiment graph, to store the meta-data and artifacts of the machine learning workloads. Let $V = \{v_i\}, i = 1, \cdots, n$ be a collection of artifacts that exist in the workload. Each artifact is either a raw dataset,

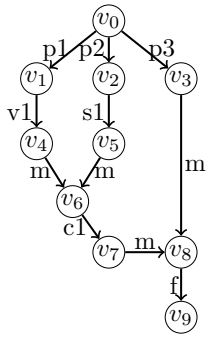
a pre-processed dataset resulting from a feature engineering operation, or a model resulting from a model training operation. Let $E = \{e_i\}, i = 1, \cdots, m$ be a collection of executed operations that exist in the workload. A directed edge e from v_i to v_j in $G(V, E)$ indicates that the artifact 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 access frequency and storage size of artifact. Every edge e has the attributes $\langle f, t \rangle$ (accessed by $e.f$ and $e.t$), which represent the execution frequency and the average run-time (in seconds) of the operation. In many cases, the vertex access frequency is equal to the execution frequency of its incoming edge. However, we make the distinction between the two in cases where users explicitly access the content of the vertex, either by printing them or saving them to disk.

Each vertex contains the data (inside the data frame) and meta-data of the artifacts, such as the name and type of the columns for datasets and name, size, the value of parameters and hyperparameters, and the error metric of the models. Each edge contains the meta-data of the operation, such as the function name, training algorithm, hyperparameters, and in some cases even the source code of the operation. When a new machine learning workload is executed, we extend the graph to capture the new operations and artifacts. If an operation already exists in the graph, we update the frequency and average run-time attributes. Otherwise, we add a new edge and vertex to the experiment graph, representing the new operation and the artifact. Similarly, if users print or save the content of an existing vertex to disk, we update the access frequency of the vertex. Figure 3a shows an example graph constructed from the code in Listing 1. To uniquely identify an edge, we utilize a hash function that receives as input the operation and its hyperparameters (if it has any).

```
1  import numpy as np
2  import pandas as pd
3
4  from sklearn import svm
5  from sklearn.feature_selection import SelectKBest
6  from sklearn.feature_extraction.text import CountVectorizer
7
8  train = pd.read_csv('../input/train.csv')
9  print train.columns # [ad_desc,ts,u_id,price,y]
10 vectorizer = CountVectorizer()
11 count_vectorized = vectorizer.fit_transform(train['ad_desc'])
12 selector = SelectKBest(k=2)
13 top_features = selector.fit_transform(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'])
```

Listing 1: Example script

Table 3b shows both the label of every edge operation, i.e., frequency and time, and the hash of the operations and their hyperparameters. Since at the time of the execution of the script, the experiment graph is empty, all the edges have an execution frequency of 1. Similarly, all the vertices, except for v_5 has access frequency of 1. v_5 which represents the *top_features* data frame in the script has a frequency of 2, since Line 15 prints its content. In order to represent operations that 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



(a)

operation	label	hash
project(ad..)	$\langle 1, 2 \rangle$	p1
project(ts, ..)	$\langle 1, 6 \rangle$	p2
project(y)	$\langle 1, 2 \rangle$	p3
vectorizer.f_t	$\langle 1, 40 \rangle$	v1
selector.f_t	$\langle 1, 60 \rangle$	s1
concat	$\langle 1, 10 \rangle$	c1
merge	$\langle 1, 0 \rangle$	m
svm.fit	$\langle 1, 100 \rangle$	f

(b)

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

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 id 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).

3.3 Workload Optimizer for Kaggle Use Case

By utilizing an experiment graph, we design a workload optimizer framework. We integrate the workload optimizer into collaborative data science platform to improve the execution of the workloads. Here, we describe the integration process of the workload optimizer for optimizing and executing kernels in the Kaggle use case (our motivating example).

Figure 4 shows the process of the workload optimization the Kaggle Infrastructure. The workflow is as follows. First, we parse a workload (i.e., a kernel) and construct the workload graph. Then, an **optimizer** component receives the workload graph and utilizes the existing experiment graph to look for optimization opportunities, namely, reusing the existing operations, warmstarting the model training, and for workloads that contain a hyperparameter tuning operation, faster hyperparameter search. The result of the optimization is another workload graph which contains precomputed artifacts, warmstarted models, and proper search space and/or an initialized Bayesian hyperparameter search process. After executing the optimized workload, we return the result to the user. After the execution, we update the experiment graph using the original workload graph (the result of step 1-Parse Workload). Finally, to ensure that we can store the experiment graph given our storage budget, we run our materialization algorithms to decide what artifacts to materialize.

It is important to note that we restrict the scope of our optimizations and materialization algorithms to one machine learning task. In Kaggle, each competition defines the machine learning task, i.e., one or multiple raw training datasets (we refer to as root datasets), a validation data set, and an evaluation function which computes the quality/error rate of the model on the validation dataset. While the experiment graph contains artifacts from all the competitions (tasks), each task corresponds to a unique connected component in the experiment graph. When optimizing a

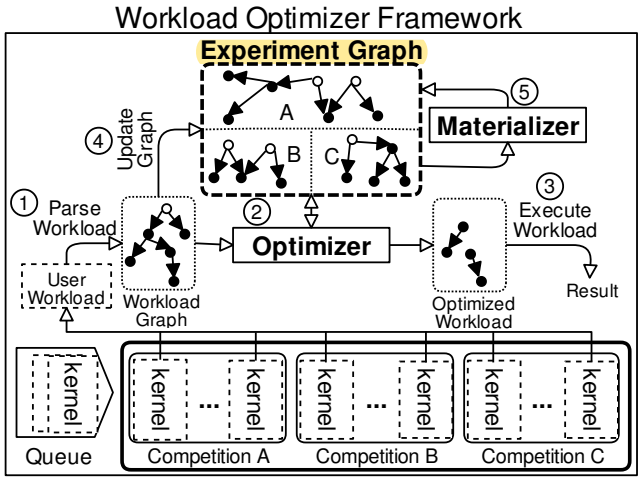


Figure 4: Improving the execution of Kaggle Kernels with Workload Optimizer

new workload, we only utilize the relevant connected component. In Figure 4, the experiment graph has three connected components representing the three competitions, A, B, and C which are rooted at 2, 2, and 1 root training datasets, depicted as hollow vertices, respectively. When optimizing a new Kaggle kernel from competition A, we only look for optimization opportunities in the connected component A of the experiment graph.

4. ARTIFACT MATERIALIZATION

Depending on the number of the executed workloads, the generated artifacts may require a large amount of storage space. For example, in the Home Credit Default Risk Kaggle competition[4], a popular script which analyzes a dataset of 160 MB, generates up to 17 GB of artifacts. Therefore, materializing every artifact under limited ~~a~~ storage budget is not feasible. In this section, we discuss 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 existing model artifacts, 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 devise a simple 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.

4.1 Materialization Problem Formulation

Bhattacherjee et al. [4] proposes an algorithm for efficient storage of different versions of a dataset (i.e., artifacts) under limited storage. The goal of the algorithm is to materialize the artifacts that result in the lowest recreation cost while ensuring the total size of the materialized artifacts does not exceed the storage capacity. However, 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 workload, i.e., materialize artifacts which result in high-quality machine learning models. We formulate the problem of materialization as a multi-objective optimization problem, with the goal of minimizing two functions given the storage requirement constraint.

Weighted Recreation Cost Function. The first function computes the weighted recreation cost of the graph:

$$WC(G) = \sum_{v \in V} (1 - v.m) \times v.f \times \sum_{e \in in\_edges(G,v)} e.t$$

where $v.m = 1$ if artifact v is materialized and 0 otherwise, $v.f$ is the access frequency of the artifact v, $in_edges(G, v)$ returns the set of edges with destination v, and $e.t$ is the execution time of the edge e. Intuitively, the weighted recreation cost indicates the total amount of execution time required to recompute all the vertices multiplied by their access 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 with access 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. The second function computes the estimated gain in quality:

$$EQ(G) = \sum_{v \in V} v.m \times estimated\_quality(v)$$

To define the function *estimated_quality*, we first define the two following sets.

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

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

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

is the set of all machine learning models to which v is connected. Now, we define the estimated quality of an artifact as:

$$estimated\_quality(v) = \sum_{m \in M(v)} m.q \times \frac{1}{1 + \mid path(v,m) \mid}$$

where $m.q$ is the quality metric of a model artifact defined as part of the task (e.g., accuracy or AUC for classification tasks and coefficient of determination for regression tasks). The multiplier $\frac{1}{1+|path(v,m)|}$ is the discount factor of the estimated quality which decreases as the distance between v and m increases. If v itself is a model, then $M(v)$ only contains v itself, as there are no outgoing edges from model artifacts, i.e., $\forall m \in M$, $estimated\_quality(m) = m.q$. When v is not a model artifact, the further v is from a model artifact the smaller the discount factor becomes, which reduces the estimated quality gained by materializing the artifact v.

Multi-Objective Optimization. Given the two functions, we would like to find the optimal set of vertices to materialize which minimizes the weighted recreation cost

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

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):

$$minimize(WC(G), \frac{1}{EQ(G)}),$$
$$\text{subject to:} \sum_{v \in V} v.m \times v.s <= \mathcal{B} \tag{1}$$

There are different strategies for solving multi-objective optimization problems [5]. However, existing solutions are time-consuming. 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.

Algorithm 1 Artifacts-Materialization

Input: $G(V, E)$ = experiment graph, root_nodes = set of root artifacts, $\mathcal{B}$ = storage budget
Output: experiment graph with materialized vertices

1: $T = 0$ ▷ current size of the materialized artifacts
2: **for** v in root_nodes **do** ▷ materialize all the root nodes
3: **if** $v.m = 0$ **then**
4: $v.m = 1$
5: $T = T + v.s$
6: **end if**
7: **end for**
8: S = empty priority queue
9: **for** v in V **do**
10: **if** $v.m = 0$ **then**
11: feasibility_ratio $= \frac{\text{feasibility}(G, v)}{v.s}$
12: insert v into S sorted by the feasibility_ratio
13: **end if**
14: **end for**
15: **for** v in S **do**
16: **if** $T + v.s \leq \mathcal{B}$ **then**
17: $v.m = 1$
18: $T = T + v.s$
19: **end if**
20: **end for**

Algorithm 1 shows the details of our method for selecting the vertices to materialize. First, we start by materializing all the root artifacts. 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 artifact, we compute the feasibility ratio, which is defined as the value of the *feasibility* function of an artifact divided by its size. Then, we start materializing all the artifacts, sorted by their feasibility ratio, until the storage budget is exhausted. The feasibility function computes the goodness of an artifact with respect to its recreation cost, how often it is used downstream, and the estimated quality gained from the artifact:

$$\begin{aligned} feasbility(G, v) = & v.out\_degree \times \\ & estimated\_quality(v) \times \\ & recreation\_cost(G, v) \end{aligned} \tag{2}$$

where $v.out\_degree$ indicates the number of outgoing edges of the artifact v, $estimated\_quality(v)$ computes the estimated quality of artifact v, and $recreation\_factor(G, v)$ indicates the weighted cost of recreating the artifact v computed as:

$$\text{recreation\_factor}(G, v) = v.f \times \sum_{e \in \bigcup_{v_0 \in roots} path(G, v_0, v)} e.t$$

, i.e., executing all the operations from the root nodes to v multiplied by the access frequency of v. Intuitively, we would like to materialize vertices which are more costly to recompute and have larger impacts on the overall quality of the experiment graph. The impact of the number of outgoing edges of an artifact is more implicit. Intuitively, an artifact with high fan-out indicates the artifact has appeared in several different machine learning workloads and pipeline. An example of these types of artifacts is cleaned and preprocessed datasets with high-quality features, where users utilize them to train several different models. Therefore, in the presence of equal estimated quality, recreation cost, and size, we are prioritizing artifacts with larger out_degrees.

4.3 Storage-Aware Artifact Materialization

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 simple 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, a 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 hash and column contents 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[5] which generates 17 GB of artifacts, deduplication result in only 8 GB of storage).

Greedy Meta-Algorithm. We propose a meta-algorithm which iteratively calls Algorithm 1 (Artifact-Materialization). We slightly modify the Artifact-Materialization algorithm, where instead of initializing the current size, T, to 0 MB on Line 1, we pass the current size as an input to the algorithm. Algorithm 2 shows the details of our storage-aware algorithm. Initially, we compute the materialized graph by calling the Artifact-Materialization algorithm and computing its compressed storage cost by calling the compress

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

method of the storage manager (Lines 1 and 2). The compress method of the storage manager removes the duplicated columns of the different artifacts of the experiment graph and returns the storage size of the graph after compression. While the size of the compressed materialized graph is smaller than the storage budget $\mathcal{B}$, we make repeated calls to the Artifact-Materialization algorithm, passing the current compressed size as the initial size argument.

Algorithm 2 Compression-Aware-Artifact -Materialization

Input: $G(V, E)$ = experiment graph, root_nodes = set of root artifacts, $\mathcal{B}$ = storage budget
Output: experiment graph with materialized vertices

```
1: G_mat = Artifact-Materialization(G, B, 0)
2: T = storage_manager.compress(G_mat)
3: while T < B do
4:     G_mat' = Artifact-Materialization(G_mat, B, T)
5:     T = storage_manager.compress(G_mat')
6:     if T ≥ B then
7:         return G_mat
8:     else
9:         G_mat = G_mat'
10:    end if
11: end while
```

Fractional Greedy Algorithm.

I have some rough ideas one what we can do here, but need to work on it a bit more. We can find all the artifacts that have common columns, and give some sort of weight to artifacts who have the highest amount of columns that are shared between other artifacts.

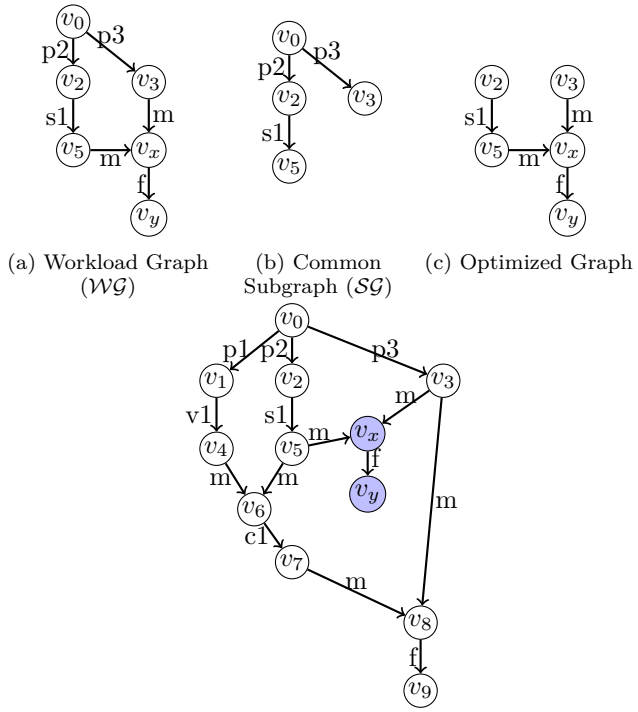
5. REUSE AND WARMSTARTING OPTIMIZATIONS

With the experiment graph constructed and materialized, we can look for optimization opportunities for feature engineering and model training operations. In this section, we propose two optimizations, namely, *Reuse* and *Warmstarting*.

5.1 Reuse ML Operations

We devise a strategy to detect overlapping operations in both the current workload and the experiment graph. If an operation already exists in the experiment graph, we directly access the resulting artifact in the experiment graph instead of executing the operation. Before executing a workload, we first transform it into its graph representation, which results in a directed graph, called $\mathcal{WG}$. Then, we traverse the experiment graph starting from the root node, using the edges of the workload graph. The result of the traversal is a subgraph of the experiment graph, which we refer to as $\mathcal{SG}$. The subgraph contains all the vertices and edges that exist in both the experiment graph and the workload graph. For every path in $\mathcal{SG}$ which originates at the root node, we return the furthest materialized vertex from the root node as the result and skip all the intermediate operations.

To demonstrate with an example, let us assume a user has executed the code in Listing 1 and Figure 3a shows the current experiment graph, where the set of materialized vertices is $\mathcal{MV} = \{v_0, v_2, v_3\}$. A new user submits the code in Listing 2. Figure 5a shows the workload graph ($\mathcal{WG}$) of



(a) Workload Graph ($\mathcal{WG}$) (b) Common Subgraph ($\mathcal{SG}$) (c) Optimized Graph

(d) Experiment graph after executing the workload (new nodes are highlighted)

Figure 5: Steps in the Reuse optimizations

the code in Listing 2. As described in the Reuse procedure, we traverse the experiment graph, starting at v_0, with the edges of $\mathcal{WG}$ which results in the common subgraph, $\mathcal{SG}$, in Figure 5b. In the subgraph $\mathcal{SG}$, the furthest vertices in each path originating at v_0 are v_5 and v_3. The Reuse procedure selects v_2 as the candidate since v_5 is not materialized. The Reuse procedure results in an optimized graph which skips operations p2 and p3 (Figure 5c).

```
8  train = pd.read_csv('../input/train.csv')
9  selector = SelectKBest(k=2)
10 top_features = selector.fit_transform(train[['ts','u_id','price']],
11                                       train['y'])
12 model = svm.SVC()
13 model.fit(top_features, train['y'])
```

Listing 2: New workload script (Imports are omitted)

5.2 Warmstart Model Training Operations

Model training operations include extra hyperparameters that must be set before the training procedure begins. Two training operations on the same data artifact using the same training algorithm could potentially have very different results based on the values of the hyperparameters. Therefore, we cannot apply the Reuse optimization in cases the hyperparameters of model training operations are different. Instead, we apply the *Warmstarting* optimization. We first need to describe the concept of model groups. For a vertex v, the model group $\mathcal{MG}_v$ refers to the set of all the machine learning models and their hyperparameters that are trained on v. If a workload contains the model training operation e_m on the vertex v_m, before executing the workload,

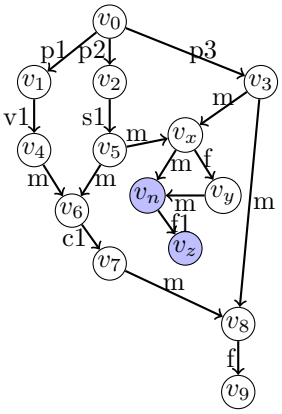


Figure 6: Experiment graph after warmstarting

we proceed as follows. First, using the same traversal strategy as explained in the Reuse procedure, we look for v_m in the experiment graph. If operation e_m also exists in the experiment graph, then the Reuse procedure will return the resulting node as the result. However, if e_m does not exist in the experiment graph, then we find the model group $\mathcal{MG}_{v_m}$. If $\mathcal{MG}_{v_m}$ is not empty, we warmstart the operation e_m with the best performing model from the model group.

To demonstrate with an example, after executing both the code in Listings 1 and 2, a new users submits the same code as in Listing 2, with a different model hyperparameters (Line 12) shown in Listing 3.

```
12 model = svm.SVC(C=0.1)
```

Listing 3: Workload with different hyperparameters

Since both models (svm.SVC(C=0.1) and svm.SVC()) are trained on the same vertex (v_x in Figure 5), the Warmstarting procedure selects the existing model node (v_y in Figure 5) to warmstart the training process.

Warmstarting can greatly reduce the total training time. However, the type of the machine learning model and the termination criteria play important roles in determining the effect of the warmstarting optimization. In the experiment section, we evaluate the effect of warmstarting on different types of models with different termination criteria.

5.2.1 *Augmenting the experiment graph*

When we utilize warmstarting, we extend the experiment graph with a merge operation which merges the dataset and the candidate model for warmstarting. The actual training operation is then applied to the merged node. As a result, we can keep track of the models that are utilized in warmstarting the training of other models which ensures reproducibility.

Figure 6 shows the experiment graph after execution of the script from Listing 3. The training operation, f1, is applied to the new vertex v_n, which is the result of merging the data artifact v_x and the model v_y. The training operation f1 has a different hash from the existing operations since the hyperparameters are different.

6. IMPROVED HYPERPARAMETER TUNING

Hyperparameters of a machine learning model greatly impact the quality of the model. There are three common techniques for tuning the hyperparameters, namely, grid search, random search, and Bayesian hyperparameter search [7, 12]. In this section, we propose three techniques to improve the process of hyperparameter tuning.

6.1 Search unpacking

Existing machine learning libraries provide black-box operations for performing hyperparameter tuning. However, the search process consists of multiple training operations that only differ in hyperparameters. Therefore, instead of using black-box operations, we consider each training run in the hyperparameter tuning process as a separate operation. This enables us to apply both the reuse and warmstarting operations to individual training operations in hyperparameter tuning.

6.2 Automatic Search Space Definition

A major challenge in all the hyperparameter tuning techniques is defining the appropriate search space. For every hyperparameter, the search space is defined as a probability density function, where values are drawn randomly according to the probability distribution. Currently, the probability density functions are defined through trial-and-error. Therefore, users may need to re-execute a hyperparameter tuning operation multiple times with different search spaces to achieve good results. Novice users, especially, suffer from this problem, as they typically lack the knowledge to define a proper search space.

Using the experiment database, we can propose search space to the users. For every model group in the experiment database, we proceed as follows. First, we extract all the hyperparameter values for the model group. Then, we estimate the distribution of every hyperparameter using density estimation methods.

I think a simple Histogram estimation should suffice here. I need to run some quick experiments to Histogram vs Kernel density estimation.

When a user executes a hyperparameter search in their workloads, we propose the estimated distributions as search space to the user.

One drawback of this approach is that a model group should have enough models with different hyperparameters, in order for the density estimation method to accurately estimate the probability distribution for the hyperparameters [11]. As a result, we can only propose a search space, when there are enough models inside a model group. In the experiment section, we evaluate the effect of the number of available hyperparameter values on the estimated distribution.

I still need to figure out how to design a meaningful experiment for Automatic Search Space definition, since there is no baseline that we can compare ourselves with. One approach would be to perform multiple hyperparameter tunings with very large budgets in a much bigger search space than the one recommended by our method. We then check how many times the best model found by the search falls in the search space that we defined.

We can also prune the search space by removing the regions that do not yield in good quality models. Since for every hyperparameter value in the experiment database, we have know the resulting model quality, we can only estimate the density for the hyperparameter values that result in a model quality greater than a threshold.

6.3 Fast Bayesian Hyperparameter Tuning

One drawback of the Bayesian hyperparameter tuning is that it requires many initial random trials until it starts to propose promising hyperparameters [7, 12]. By utilizing the experiment database, we devise a strategy to decrease the number of initial trials (or in some cases avoid it altogether). As a result, when users perform bayesian hyperparameter tuning in their workload, they receive promising hyperparameters faster.

Our solution is as follows. For every existing model group in the experiment database, we start a Bayesian tuning process. A model group is a set of models that only differ in their hyperparameter values. We then initialize the search process by including the existing hyperparameters and the corresponding model quality from the model group.

When users submit a workload, there are two possible scenarios. In the first scenario, the user defines a model training operation in the workload but does not specify any hyperparameters and requests promising hyperparameter values. In this case, we first find the model group that this workload belongs to. Then, from the corresponding Bayesian tuning process, we get the proposed hyperparameters to train the model. In the second scenario, a user may specify a Bayesian hyperparameter tuning process with a specific budget inside their workload (we call this a user-defined Bayesian hyperparameter tuning). In this scenario, similar to the first scenario, we find the corresponding Bayesian tuning process and utilize it as a starting point for the user-defined Bayesian hyperparameter tuning process. As a result, the user-defined Bayesian hyperparameter tuning immediately starts to propose promising hyperparameters as it does not need to perform the initial random trials.

7. EVALUATION

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

7.1 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][6]. Table 2 shows the id, components, and number of executions of the OpenML pipelines.[7]

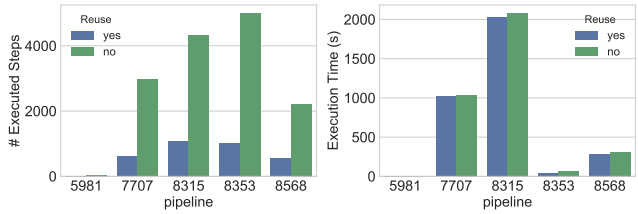
Kaggle workloads: In the second set of workloads, we target interactive machine learning loads where a large portion of the analysis is spent on data and feature preprocessing. We design the workload by studying several Kaggle scripts and populating the database based on those. Here's the description of the Kaggle challenges that we use.

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

[7]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 2: OpenML pipeline descriptions.



(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

This is my next immediate ToDo.

7.2 Reuse and Warmstarting for the OpenML Workload

7.2.1 *Feature Processing Reuse*

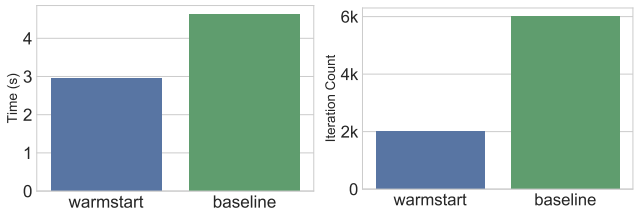
In Figure 7, we show the effect of the reuse optimization on the OpenML workflow. The number of executed steps drastically decreases as the majority of the pipelines have the exact same data transformation steps and they only differ in the hyperparameters of the model (Figure 7a). However, Figure 7b 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

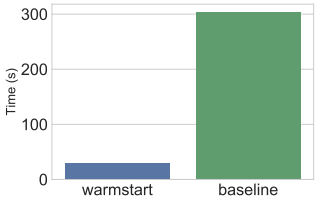
7.2.2 *Model Warmstarting*

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



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

7.2.3 *Combined Optimization*

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

7.3 Evaluation of the Improved Hyperparameter Tuning

7.3.1 *Search Space Proposal*

7.3.2 *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[8], 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

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

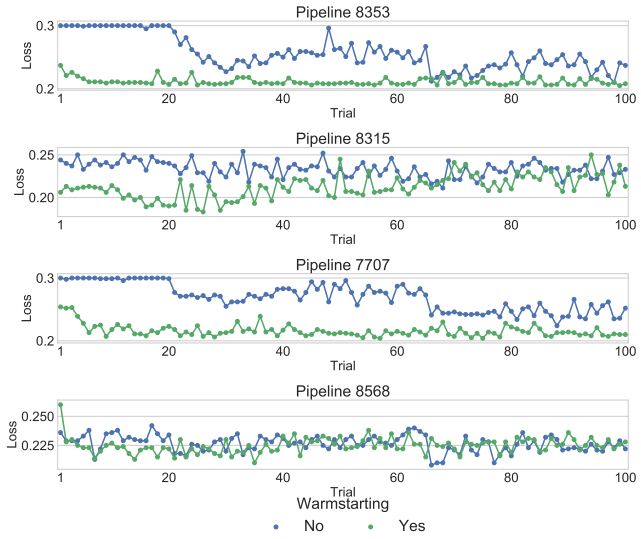


Figure 9: Loss value of 100 Trials with and without warmstarting

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

8. RELATED WORK

- repository of ml experiments
- Hyperparameter tuning, random and grid search
- Dataset versioning and materialization [4, 15]
- Transfer learning

9. CONCLUSIONS

10. REFERENCES

[1] Google colaboratory. `https://colab.research.google.com`.

[2] Kaggle. `https://www.kaggle.com`.

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

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

[5] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.

[6] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.

[7] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[9] H. Miao and A. Deshpande. Provdb: Provenance-enabled lifecycle management of collaborative data analysis workflows. *Data Engineering*, page 26, 2018.

[10] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*, 2017.

[11] B. W. Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[12] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[13] J. Vanschoren, H. Blockeel, B. Pfahringer, and G. Holmes. Experiment databases. *Machine Learning*, 87(2):127–158, May 2012.

[14] J. Vanschoren, J. N. Van Rijn, B. Bischl, and L. Torgo. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014.

[15] M. Vartak, J. M. F da Trindade, S. Madden, and M. Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1285–1300. ACM, 2018.

[16] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Model db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.