Optimization of Machine Learning Workloads with Experiment Databases

Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl

DFKI GmbH Technische Universität Berlin
{behrouz.derakhshan, alireza.rm, tilmann.rabl, volker.markl}@dfki.de

ABSTRACT

Machine learning workloads have varying characteristics. Some involve a large user base where a combination of experts and novice users are trying to design machine learning pipelines and execute them on specific tasks, such as online education, data science challenges. Some involve fewer users, typically experts, working together to solve a task. For example, a team of data scientists in a company trying to design a recommender system based on the available training data. Both workloads are interactive and require many iterations to improve the solution. In such scenarios, communication between the users involved is not optimal and as a result, many repetitions may occur. Repetitions can be of the form of repeated data preprocessing, hyperparameter search, and model training.

Using experiment databases, where a log of previous machine learning experiments is stored, we propose a solution that utilizes the information in the experiment database to improve the process of design and execution of machine learning workloads. Specifically, we utilize the logs in the experiment databases to reduce the data processing and model training time by caching and reusing the preprocessed data and trained models. Moreover, we leverage the logs to enhance the hyperparameter optimization process and provide the users (both expert and novice) with better hyperparameter settings in a shorter amount of time.

PVLDB Reference Format:
Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. Optimization of Machine Learning Workloads with Experiment Databases. PVLDB, xx(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Machine learning is at the core of academic and industry. To make sense of the data, one must design and execute a machine learning pipeline which consists of a set of data transformation steps and a model building step. Furthermore, for each step of the pipeline (data transformation or model building), one has to set several hyperparameters. The space of the available tools, the data preprocessing methods, the training algorithms, and their hyperparameters is extremely large. This overwhelms expert data scientist, let alone novice users.

To improve the process of designing and executing machine learning pipelines, the scientific community has started to employ two collaborative approaches. The goal of these approaches is to increase the collaboration between data scientists, ease reproducibility of experiments, ease teaching data science, and provide an intuitive way to share results. In the first approach, data science platforms, such as Data World[1] and Google colabratory[2] provide an intuitive way of sharing scripts and allow execution of the scripts on the platform using Jupyter notebooks [5]. Users can view other users' scripts and learn how they are solving the same or a similar problem. Moreover, users can immediately execute other users' scripts to check the results for themselves or modify the scripts to include their own custom code. This approach allows users to share results and ensures the reproducibility of the experiments. The popular data science competition platform, Kaggle, utilizes this approach to allow competitors to share ideas and scripts, which ultimately lead to better solutions for the given data problem. Online education platforms such as Coursera also utilize the same approach to enable hands-on practice for data science courses. The second approach is more systematic. In this approach, a tool stores logs of machine learning experiments which includes information about the data, the pipeline components, the training algorithms, the hyperparameters, and the evaluation results in a database, typically referred to as an experiment database [10]. Data scientists can then query other users' experiments to search for more detailed information such as the types of pipeline components (data transformations and models), hyperparameters, and evaluation for specific data problem. Examples of these systems are OpenML [11], ModelDB [13], and ProvDB [6].

Data analytics and machine learning solutions are the results of many iterations of trial-and-error, where one or more users analyze a collection of datasets and apply different transformations to form different hypotheses. Based on these hypotheses, users design data processing pipelines and training algorithms to process the datasets and train a model. This process continues until the solution is satisfactory. While storing these solutions in a structured manner inside the experiment databases provide valuable insights,

Proceedings of the VLDB Endowment, Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

[1]https://data.world
[2]https://colab.research.google.com

there are two problems with it. First, the final solution only represents a fraction of the executed operations. For example, a user typically performs several hours of exploratory data analysis only to gain insight into the problem she is trying to solve, before actually designing the final pipelines. However, most of the existing experiment databases treat these initial analysis as non-essential and only store the final result. Second, even without storing the non-essential operations, the amount of information available in the database is typically too much to digest. Therefore, querying the experiment database may create more overhead than just simply running a new experiment, regardless of whether or not some parts of the experiments are repeated.

Contrary to the existing work, we propose to store every executed operation and the resulting artifacts (we define artifacts as any raw, preprocessed, and aggregated data and machine learning models). Furthermore, we automatically extract information from the experiment database to optimize the process of design and execution of the future machine learning workloads. We limit the scope of our optimizations to the workloads belonging to the same task. A task describes what the goal of the machine learning workloads should be (for example, train a classification model on a training dataset that maximizes the F_1[3] score on a given evaluation dataset). While our solution is applicable to multiple tasks, the information gained from one task does not affect the optimizations applied to other tasks.

In order to extract the necessary information from the experiment database with ease, we represent the data inside the experiment database using a graph (called the experiment graph). In the graph, data artifacts are represented as vertices and operations that map one artifact to the other as edges.

Our goal is to leverage the experiment graph to speed up the design and execution of machine learning pipelines. Our solution comprises of two parts. In the first part, based on ~~the~~ existing workloads in the experiment database, we devise a strategy to materialize artifacts. In many real-world scenarios, the size of the generated artifacts may increase to the orders of 100s of Gigabytes or Terabytes. Therefore, under a limited storage capacity, the materialization strategy should materialize the artifacts that have a high probability of reuse in future workloads.

In the second part, given the materialized experiment graph, when presented with a new workload, we detect whether the existing meta-data in the experiment graph and the materialized artifacts can be utilized to optimize the design and execution of the new workload. Concretely, we utilize the information to optimize the workload design and execution in three ways. We propose three main optimizations, namely, reuse, warmstarting, and improved hyperparameter tuning. In reuse, we look for opportunities to reuse an existing materialized artifact and avoid data reprocessing. Reuse decrease the total data processing time and can enable interactive data analysis, especially during the exploratory data analysis that we expect many repetitions of operations. In warmstarting, we devise a method to detect if we can warmstart the model in the workload with an existing materialized model artifact. Warmstarting reduces the convergence time, resulting in faster training time and in some case in a model with better quality. 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 [4, 9]. 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 process. As a result, non-expert users struggle with setting an effective search space. We utilize the meta-data in the experiment database to define (or propose) promising search space to the users. Moreover, after setting the search space, Bayesian hyperparameter tuning techniques require many initial trials until they start to propose promising hyperparameters. We utilize the meta-data in the experiment database to initialize the Bayesian hyperparameter tuning process to skip the trials. As a result, we the Bayesian hyperparameter tuning process proposes promising hyperparameters faster.

In summary, our contributions are:

- A system that optimizes the process of design and execution of machine learning worklaods by utilizing an experiment database
- An algorithm for materializing artifacts in the experiment graph under limited storage capacity
- Enabling reuse of data operation, which enables interactive data analysis, especially during the exploratory data analysis phase
- Enabling the warmstarting optimization, which decreases the total model training time and in some cases, may lead to models with higher quality
- Increase in the efficiency of hyperparameter tuning by incorporating the information from the experiment database, which leads to more promising hyperparameter values

The rest of this document is organized as follows.

2. BACKGROUND

In this section, we first provide an example use case and outline its deficiencies. Next, we describe experiment databases. Finally, we discuss how we improve the example use case by incorporating experiment databases.

2.1 Example Use Case

The data science competition platform, Kaggle, enables organizations to host data science challenges. Users participate in the challenges, write solutions, and submit these solutions. 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 workload publicly available to other users. As a result, the entire community works together to find high-quality solutions.

Kaggle utilizes docker containers to provide isolated computational environments (called Kaggle kernels). 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[4]). In busy times, this results

[3]https://en.wikipedia.org/wiki/F1_score

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

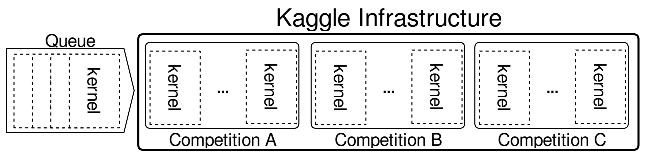


Figure 1: Kaggle Infrastructure

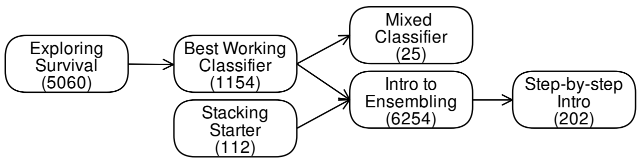

Exploring Survival (5060)
Best Working Classifier (1154)
Stacking Starter (112)
Mixed Classifier (25)
Intro to Ensembling (6254)
Step-by-step Intro (202)

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

in users to be placed in queues (especially for GPU-enabled machines) until resources become available. Figure 1 shows the infrastructure of Kaggle. Kaggle groups kernels by competition. When users request to execute a kernel, if there are enough resources available, their kernel is executed. However, when enough resources are not available, the kernel is inserted into a queue and only executed when existing kernels are completely executed.

Every user ~~that~~ 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 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[5], 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 cites the kernel Exploring Survival as his/her inspiration. The numbers show how many times other users copy a kernel into their own 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. By using isolated docker containers, the Kaggle platform cannot detect such similarities and must re-execute every operation regardless of how many time it is executed before.

2.2 Experiment Database

Experiment databases include data and meta-data of different data analytics and machine learning experiments executed over time [6, 11, 7, 13]. They include different information about datasets, data processing pipelines, machine learning models, execution of machine learning training, and quality of the models. Moreover, experiment databases can also store the artifacts generated during the execution of a workload, such as datasets, intermediate data (resulting

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

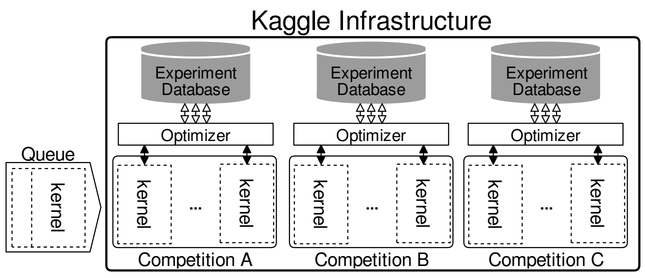


Figure 3: Utilizing Experiment Databases in the infrastructure of Kaggle

from applying data transformation operations), and machine learning models and their parameters.

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.

2.3 Improved Use Case

By utilizing an experiment database, we plan to optimize a new workload by automatically removing redundant operations, speed up the model training, and perform more efficient hyperparameter tuning.

Figure 3 shows how we utilize the experiment database in the Kaggle Infrastructure. The workflow is as follows. Before executing a kernel (workload), first, an optimizer component analyzes the workload. Then, the optimizer searches for optimization opportunities by querying the experiment database. For example, if a new kernel is trying to standardize the dataset or perform a model training, the optimizer queries the experiment database to find out if other users have already executed the operations. Based on the result of the query, the optimizer then decides whether to retrieve the standardized data or the model from the experiment database or execute the operations in the kernel.

In the next sections, we first describe how we utilize the experiment database and what types of optimizations an experiment database enables us to perform.

3. MACHINE LEARNING WORKLOADS

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 database. We also introduce a materialization strategy for only storing the generated data artifacts during the execution of a workload. Lastly, we discuss how do we utilize the experiment database to optimize new workloads.

3.1 Operations in ML Workloads

We assume the main units of work are dataframe (e.g., Pandas, R Data Frames, and Spark DataFrames) like objects that contain one or many columns, where all the data items

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

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 dataframe 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*

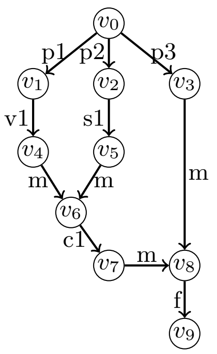
Model training operations are a group of operations that receive a dataset (or one or multiple columns of a dataset) and return a machine learning model. The result of model training operations can either be used in other data and feature engineering operations (e.g., applying PCA to reduce the number 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 machine learning models with the best performance. A hyperparameter tuning operation is defined by a budget and a search method. The budget specifies how many models with different hyperparameter values the operation should train and the search method specifies what search strategy should be incorporated. We limit our focus to popular search methods, namely, grid search, random search, and Bayesian hyperparameter search [1, 9].

3.2 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 $\mathcal{V} = \{v_i\}, i = 1, \cdots, n$ 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 $\mathcal{E} = \{e_i\}, i = 1, \cdots, m$ be a collection of executed operations that exist in the workload. A directed edge e from v_i to v_j in $\mathcal{G}(\mathcal{V}, \mathcal{E})$ indicates that the artifact



(a)

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

(b)

Figure 4: Experiment graph constructed from the Listing 1 (a) and the hash of the operations in the scripts (b)

v_j is (fully or partially) derived from the artifact v_i by applying the operation in e. Every vertex v has the attribute $\langle s\rangle$ (accessed by $v.s$), which represents the storage size of artifact when materialized. Every edge e has the attributes $\langle f, t\rangle$ (accessed by $e.f$ and $e.t$), where f represents the frequency of the operation (the number of times the operation has been executed) and t represents the average run-time (in seconds) of the operation. Each vertex contains meta-data about 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 about 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. Figure 4a 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 hyperparameters.

```
import numpy as np
import pandas as pd

from sklearn import svm
from sklearn.feature_selection import SelectKBest
from sklearn.feature_extraction.text import CountVectorizer

train = pd.read_csv('../input/train.csv')
print train.columns # [ad_desc,ts,u_id,price,y]
vectorizer = CountVectorizer()
count_vectorized = vectorizer.fit_transform(train['ad_desc'])
selector = SelectKBest(k=2)
top_features = selector.fit_transform(train[['ts','u_id','price']],
                                      train['y'])
X = pd.concat([count_vectorized,top_features], axis = 1)
model = svm.SVC()
model.fit(X, train['y'])
```

Listing 1: Example script

Table 4b shows both the label of every edge operation, i.e., frequency and time, and the hash of the operations and their hyperparameters. We assume the operations project(ad..) and vectorizer.f_t already exist in the experiment graph. Therefore, when adding the new operations, we must update their frequency to 2. In order to represent operations that process multiple input artifacts, e.g., concat and model.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 (run-time of 0 seconds). Then, we draw an edge from the merged vertex which represents the actual operation. For example, in Figure 4a, before applying the concatenation operation, we merge v_4 and v_5 into v_6, then we apply the concatenation operation (c1). This is a critical step for our materialization algorithm in Section 3.3 and our optimization strategies discussed in Sections 4 and 5.

It is important to note that based our definition of a task in Section 1 ~~we~~ the workloads belong to multiple different tasks, the constructed graph will contain one connected component for every task. In the next sections, we assume that the experiment graph contains information about 1 task, although, all the methods described can be applied to multiple tasks as well.

3.3 Materialization of the Experiment Graph

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[6], one of the popular scripts that analyzes a dataset of 150 MB, generates up to 10 GB of artifacts.

10 GB is a rough estimate, I need to finish the code to accurately calculate the number

Therefore, materializing every artifact is not feasible. In this section, we discuss our algorithm for materializing a subset of the artifacts under limited storage, a modified version of the algorithm presented by bhattacherjee et al. [2]. The goal of the algorithm is to materialize the artifacts that result in the lowest weighted recreation cost while ensuring the total size of the materialized artifacts does not exceed the storage capacity. We define the weighted recreation cost of the graph $\mathcal{G}$ (WC), given the set of materialized vertices $\mathcal{MV}$ as

$$WC(\mathcal{G}, \mathcal{MV}) = \sum_{e \in \{e' \in \mathcal{E} | dest(e') \notin \mathcal{MV}\}} e.f \times e.t$$

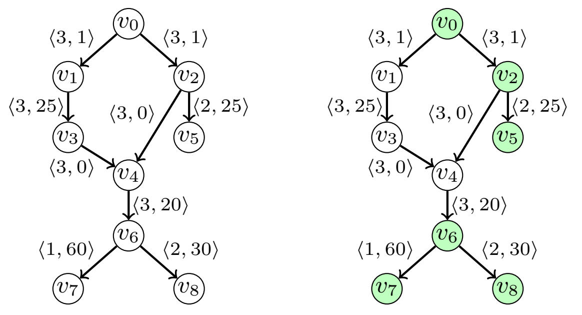
where $dest(e)$ represent the destination vertex of the edge e. The weighted recreation cost indicates how much time do we need to spend to execute all the operations in the graph, since the beginning of time. For un-materialized artifacts, we must consider the frequency of the operations that produce the artifact. For example, in Figure 4, if we do not materialize v_4 and the operation vectorizer.f_t has a frequency of 2, we must consider both executions of the operation when computing the weighted cost. Whereas, if v_4 is materialized, the vectorizer.f_t operation has no impact on the weighted recreation cost.

Algorithm 1 shows the details of our method for selecting the vertices to materialize. First, we start by initializing the materialized vertices set ($\mathcal{MV}$) to contain the root artifact (v_0) which represents the raw dataset. This is essential as many of the feature engineering and model building operations are not invertible. As a result, we cannot reconstruct the raw dataset if it is not materialized. Then, while the storage limit is not reached, we materialize vertices with the

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

Algorithm 1 Materialization of Artifacts

```
Input: $\mathcal{G}(\mathcal{V}, \mathcal{E})$ = experiment graph, $\mathcal{B}$ = storage limit
Output: $\mathcal{MV}$ = set of materialized artifacts
1: $T = v_0.size$, $\mathcal{MV} = \{v_0\}$
2: do
3:     $\mathcal{CV} = \{v \in \mathcal{V} | v \notin \mathcal{MV}, T + v.s \leq \mathcal{B}\}$
4:     $v^* = \operatorname{argmax}_{v \in \mathcal{CV}} \frac{\rho(\mathcal{G}, v)}{v.s}$
5:     $\mathcal{MV} = \mathcal{MV} \cup \{v^*\}$
6:     $T = T + v^*.size$
7: while $\mathcal{CV} \neq \emptyset$
```



(a) Original Graph (b) Materialized Graph

vertex	$\boldsymbol{v_0}$	v_1	$\boldsymbol{v_2}$	v_3	v_4	$\boldsymbol{v_5}$	$\boldsymbol{v_6}$	$\boldsymbol{v_7}$	$\boldsymbol{v_8}$
size (MB)	10	8	2	40	42	1	30	2	3
$\boldsymbol{\rho}$ (s)	—	3	3	78	81	52	141	107	154
ratio	—	0.37	1.5	1.95	1.93	52	4.7	53.5	51.3

(c) List of vertices, their sizes, recreation costs, and the cost over size ratio (Bold vertices are materialized).

Figure 5: Artifact materialization based on Algorithm 1 when storage capacity is 55 (MB)

maximum value of weighted recreation cost over size. We compute the weighted recreation cost (ρ) of the vertex v as,

$$\rho(\mathcal{G}, v) = \alpha(\mathcal{G}, v) \times \sum_{e \in path(\mathcal{G}, v_0, v)} e.t$$

where $\alpha(\mathcal{G}, v)$ represents the access frequency of the vertex v which is the same as frequency of the edge (or any of the edges in case of merge operation) connected to v. The set $path(\mathcal{G}, v_0, v)$ represents the set of all edges from the root node to the vertex v. For example, in Figure 5a, the recreation cost of v_4 is $3 \times (0+0+25+1+1) = 81$. The ratio of the weighted recreation cost over size has the unit second per megabyte. For example, the ratio 10 s/mb for an artifact, indicates that we need to spend 10 seconds to recreate 1 megabyte of the artifact. Figure 5 shows an example of the materialization process when the storage capacity is 55. For v_0 we do not compute ρ as v_0 is always materialized.

3.3.1 *The Effect of Model Quality on the Materialization Decision*

For this I need to design experiments to check if it has a positive impact. If we don't have space, then this will be moved to future work.

Since the goal of all the workloads in the experiment graph is to solve the same task (as described in 1), all the machine learning models will be evaluated using the same quality metric. Therefore, we can utilize the quality of the model in

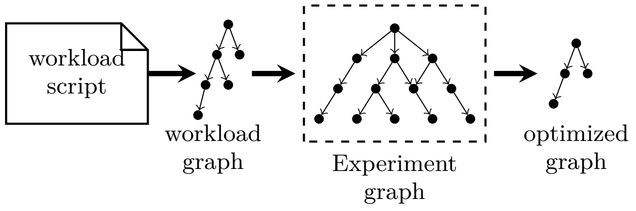


Figure 6: System workflow

the materialization algorithm. We propose a simple method for utilizing the model quality in the materialization decision algorithm. We start by adding a new attribute, q, to every edge. The new attribute is computed as follows. If the edge e belongs to no path that leads to a predictive model, we assign q to the average quality of all the predictive models in the experiment graph. If e belongs to only one path that leads to a predictive model, then we assign q to the quality of the model. If e belongs to multiple paths that lead to different predictive models, we assign q to the quality of the model with the maximum quality among all the models. After computing q, we include it in the computation of ρ, by multiplying $e.q$ by $e.t$ in the summation.

3.4 Optimization Workflow

Figure 6 shows the workflow of our system. First, we transform the workload into its graph representation. Then we utilize the experiment graph to optimize the workload using the techniques in Sections 4 and 5. This results in a new workload graph which depending on the types of optimizations may have a fewer number of operations, faster operations, or operation configurations that lead to higher quality machine learning models.

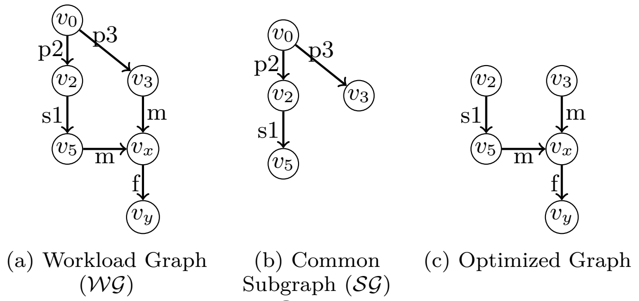
4. REUSE AND WARMSTARTING OPTIMIZATIONS

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, Warmstarting.

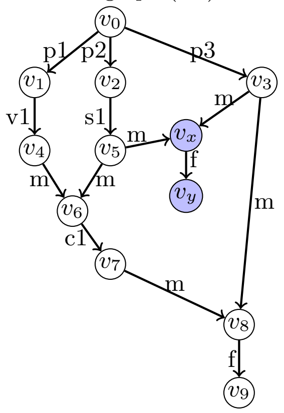
4.1 Reuse Optimization ML Operations

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 4a shows the current experiment graph, where the set of materialized



(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 7: Steps in the Reuse optimizations

vertices is $\mathcal{MV} = \{v_0, v_2, v_3\}$. A new user submits the code in Listing 2. Figure 7a shows the workload graph ($\mathcal{WG}$) of 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 7b. 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 that skips operations p2 and p3 ~~are skipped~~ (Figure 7c).

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

Listing 2: New workload script (Imports are omitted)

4.2 Warmstarting Optimization For 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. We refer to the set of every machine learning model trained on the same data artifact but only differs in hyperparameters, as a model

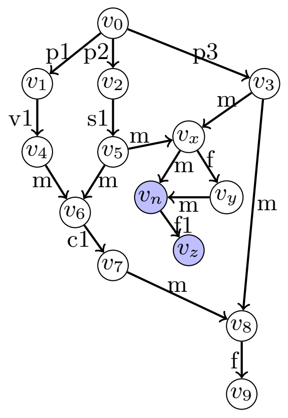


Figure 8: Experiment graph after warmstarting

group. If a workload contains a model training operation, e_m, on a vertex, v_m, before executing the workload, we proceed as follows. First, using the same traversal strategy as explained in the Reuse procedure, we look for v_t in the experiment graph. If v_t is in the experiment graph, then, we find the model group containing all the models trained on v_t. Finally, if the model group is not empty, we warmstart the operation e_t 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 7), the Warmstarting procedure selects the existing model node (v_y in Figure 7) 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.

4.2.1 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 8 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.

5. 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 [4, 9]. In this section, we propose three techniques to improve the process of hyperparameter tuning.

5.1 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.

5.2 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 [8]. 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.

5.3 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 [4, 9]. 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.

6. EVALUATION

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 [3][7]. Table 2 shows the id, components, and number of executions of the OpenML pipelines.[8]

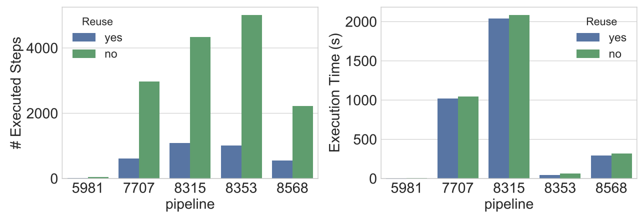
Kaggle workloads: In the second set of workloads, we target interactive machine learning loads where a large portion of the analysis is spent on data and feature preprocessing. We design the workload by studying several Kaggle scripts

[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 2: OpenML pipeline descriptions.



(a) Executed Steps (b) Execution Time

Figure 9: Effect of the reuse optimization on the total number of executed transformations and the total execution time for every OpenML pipeline

and populating the database based on those. Here's the description of the Kaggle challenges that we use.

This is my next immediate ToDo.

6.2 Reuse and Warmstarting for the OpenML Workload

6.2.1 *Feature Processing Reuse*

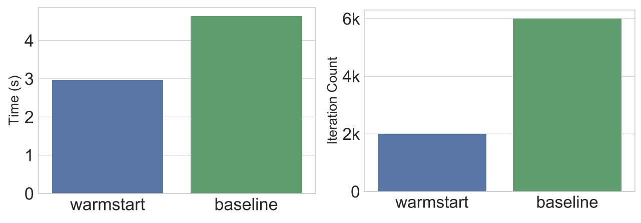
In Figure 9, we show the effect of the reuse optimization on the OpenML workflow. The number of executed steps drastically decreases as the majority of the pipelines have the exact same data transformation steps and they only differ in the hyperparameters of the model (Figure 9a). However, Figure 9b 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

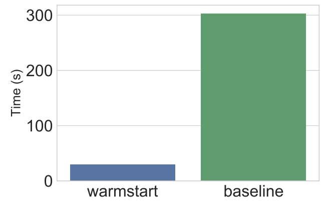
6.2.2 *Model Warmstarting*

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 10 shows the result of the model warmstarting optimization on two types of models in the experiment database. Figures 10a and 10b 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 10: 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 10c 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).

6.2.3 *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[9], classifying customers as good or bad credit risks using the German Credit data from the UCI repository [3]. 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 11 shows the average of losses of the 100 trials for the 10 experiments. Warm starting the search decreases the overall loss of the trials.

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

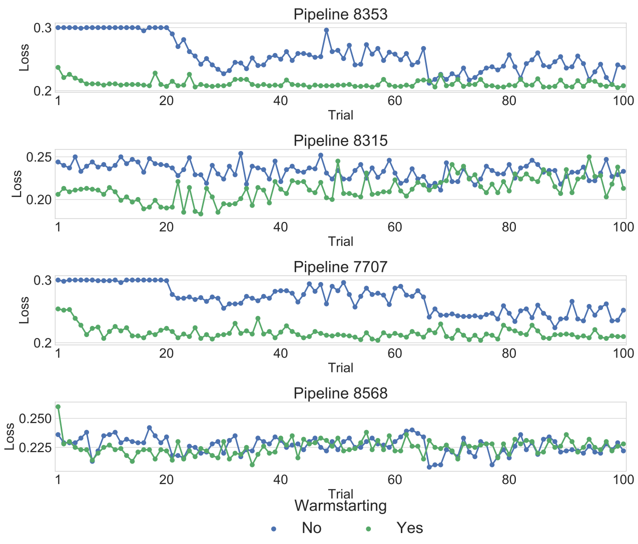


Figure 11: Loss value of 100 Trials with and without warmstarting

7. RELATED WORK

- repository of ml experiments
- Hyperparameter tuning, random and grid search
- Dataset versioning and materialization [2, 12]
- Transfer learning

8. CONCLUSIONS

9. REFERENCES

[1] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(Feb):281–305, 2012.
[2] 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.
[3] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.
[4] 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.
[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.