Demonstration Paper

Hyppo: Efficient Discovery and Execution of Data Science Pipelines in Collaborative Environments

Antonios Kontaxakis
antonios.kontaxakis@ulb.be
Université Libre de Bruxelles
Brussels, Belgium
Universitat Politècnica de Catalunya
Barcelona, Spain

Dimitris Sacharidis
dimitris.sacharidis@ulb.be
Université Libre de Bruxelles
Brussels, Belgium

Alkis Simitsis
alkis@athenarc.gr
Athena Research Center
Athens, Greece

Alberto Abelló
aberto.abello@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Sergi Nadal
sergi.nadal@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

ABSTRACT

In exploratory data science and machine learning (ML), developing an effective and efficient solution involves the exploration of numerous pipelines per dataset, considering various combinations of data preprocessing, feature selection, model selection, evaluation metrics, etc. This is an iterative process of trial and error, where a pipeline is revised and refined until a satisfactory level of performance is achieved. Such a process can be substantially improved by leveraging collaboration opportunities and automated pipeline optimization. We demonstrate Hyppo, a scalable collaboration system designed for interactive discovery and automated optimization, resulting in a system that supports evaluating and sharing pipelines with minimal overhead. Hyppo takes advantage of numerous optimization possibilities as exploiting equivalences among tasks, sharing computations, artifact materialization, and reuse from past executions to derive better execution plans. We showcase Hyppo with open data and pipelines from popular public code repositories and catalogs.

Demo video: https://youtu.be/iilPV_xtXwM

1 INTRODUCTION

Teams of data scientists exchange and refine ideas, striving to identify the right data processes for optimal performance. Platforms like Kaggle offer global collaboration through Jupyter Notebooks, providing spaces where scientists can share scripts and results. Similarly, OpenML and Hugging Face further support collaboration by enabling the storing of essential elements like ML pipelines, models, and analytical results, facilitating convenient resource exchanges. These platforms have revolutionized how data scientists work by enabling the convenient sharing of scripts, models, and results. Their role as execution engines and repositories for data science artifacts is crucial in fostering a collaborative and accessible environment.

Although beneficial, such an approach introduces inherent challenges and overheads. Due to the large variety of alternatives, data scientists tend to spend significant time reviewing and searching previously proposed solutions whilst in turn, the refinement process creates multiple re-executions of partial results, introducing unnecessary resource consumption. Additionally, scientists have to explore large artifact repositories, seeking alternative solutions to implement their tasks. Hence, exploring and reusing previously computed results largely remains a manual process, which increases the time, effort, and resources required for artifact exploration. This inefficiency often leads to redundant computations and limits the ability to leverage prior knowledge effectively. Hence, it is essential for the next generation of data science collaboration tools to aid users in discovering a manageable (handful) number of relevant high-performing pipelines and provide automated optimization.

Related work. Current techniques to address the challenges of exploratory data science and ML include approaches that automatically select which pipelines to investigate next, thereby reducing the *effort* required by scientists (e.g., AutoML [2]). Additionally, several systems [1, 4, 8, 9] have been developed that make use of various optimization techniques such as reuse and materialization, seeking to reduce the execution *cost* of pipelines. To date, no system has been developed to address both challenges simultaneously, although all the aforementioned techniques aim to enhance the efficiency and effectiveness of exploratory data science.

Contribution. In this paper, we demonstrate Hyppo via interactive Notebooks. Hyppo operates as a middleware between the user and the execution engine that (a) streamlines the search for pipelines by storing previous executions and useful metadata as a hypergraph, called History, and (b) optimizes resource usage, reducing time, effort, and cost required for exploring alternative solutions by solving a path discovery problem over the History. Hyppo achieves two goals: pipeline *optimization* (as described in [3]) and pipeline *discovery* features (an extension to [3]). In this version, we also enable ad hoc exploration by implementing History on top of a graph database and allow users to directly query History via a Python API that exposes several key functions for discovery.

Attendees will be able to discover, design, and evaluate pipelines for a set of predefined real-world tasks and datasets. The demo's purpose is threefold: (a) highlight that users can navigate and review a rich history of previously executed pipelines in a few steps with little effort; (b) showcase that users can easily plug their own operators, which Hyppo picks up and uses to generate better execution plans; and (c) emphasize the benefits of Hyppo's plethora of automated optimization capabilities. By integrating both discovery and optimization capabilities, Hyppo reduces the burden on data scientists and enables more efficient exploration.

```
(1) Discovery
1 hyppo = Hyppo("catalog_path")
2 dataset = "HIGGS"
3 hyppo.best_metrics(dataset, num=3)

metric  low_score  high_score
  'F1'       0.58        0.88
   'P'       0.40        0.72
   'R'       0.61        0.90

4 hyppo.retrieve_best_pipeline(dataset, "F1", num=1)

[('scaler', StandardScaler()),
 ('pca', PCA(n_components=3)),
 ('svm', SupportVectorMachine()),
 ('F1', F1ScoreCalculator())]

5 hyppo.popular_operators(dataset, "Classifier")

[(42, 'rfc'), (23, 'svm'), (4, 'ridge')]

6 hyppo.view_dictionary('rfc')

{'rfc': ['sk.ensemble.RandomForestClassifier',
         'tfdf.keras.RandomForestModel',
         'cuml.ensemble.RandomForestClassifier']}

(2) Optimization
7 pipe = Pipeline([('scaler', StandardScaler()), ('pca',
     PCA(n_components=3)), ('rfc', cuml.ensemble.
     RandomForestClassifier()), ('F1',
     F1ScoreCalculator())], dataset)
8 pipe.run()

{'time': 1200ms, 'F1': 0.91}

9 opt_pipe = hyppo.optimal_plan(pipe)
10 opt_pipe.run()

{'time': 700ms, 'F1': 0.91}
```

Snippet 1: An example of Hyppo's Python API

2 MAIN DEMONSTRATION SCENARIO

Hyppo enables data scientists (1) to *discover* previously executed pipelines and receive insights, and (2) to *optimize* the execution of a pipeline by exploiting historical information and equivalences among operators and artifacts. We showcase these functionalities with our main demonstration scenario. We defer the discussion on Hyppo's system components and how they work to Section 3.

We consider Kaggle as an illustrative example of online collaborative development in data science. There, several competitions of varying complexity are available. Here, we use the Higgs Boson Machine Learning Challenge,[1] a complex task in the field of particle physics, specified as a binary classification problem.

Snippet 1 illustrates how Hyppo's Python API aids data scientists in (1) *discovery*, e.g., exploring the pipelines already implemented, obtaining insights from the best performing, and (2) *optimization* of new pipeline executions. The snippet is presented as a notebook, alternating between code and output, and comprises two phases.

In the first phase, *discovery*, the data scientist wishes to draw insights on how to design a pipeline to solve the challenge. They first review evaluation metrics achieved on the specific dataset (Lines 2–3) and then decide to retrieve the top-performing pipeline according to the F1 score (Line 4). By inspecting this pipeline, the data scientist develops an intuition on what preprocessing steps appear to work well (e.g., normalization with `StandardScaler`, followed by extracting the three principal components with `PCA`). Thus, they decide to keep these preprocessing steps and explore alternate classification models. Then, the data scientist would like to know what other models have been used for this task. So, they query Hyppo for popular operators of type `Classifier` that have been used in pipelines involving the `HIGGS` dataset (Line 5). It appears that random forest classifier `rfc` is very popular. Hence, the data scientist looks up the available implementations of `rfc` (Line 6). At this point, the data scientist is ready to define their pipeline to be executed.

In the second phase, *optimization*, the data scientist interacts with Hyppo in order to efficiently execute their pipeline. First, they define the pipeline as a sequence of tasks over a dataset. Specifically, the data scientist copies the preprocessing steps of the best-performing pipeline so far and appends a random forest classifier as the final step (Line 7). They can now execute the pipeline as is (Line 8). The pipeline achieves a better evaluation metric, and the data scientist is happy. However, they have not exploited the full capacity of Hyppo, which is optimizing the execution of pipelines. Specifically, the data scientist can give the pipeline as input to the optimizer of Hyppo, which returns an optimal execution plan (Line 9). Executing that plan (Line 10) results in a considerable decrease in execution time.

3 OVERVIEW OF THE HYPPO SYSTEM

Next, we present the Hyppo system at a high level, suitable for understanding its API and the demonstration scenaria; a detailed description of Hyppo internals can be found in [3].

Pipelines. A pipeline is a collection of computational *tasks* that produce and consume *artifacts*. In practice, ML tasks are multi-input and multi-output. Directed graphs, used in prior work [1, 8], cannot distinguish between multi-input tasks (e.g., joins), which require *all* inputs and are connected with 'AND' edges, and alternative tasks, which require *any* input and are connected with 'OR' edges. Hyppo employs a novel pipeline representation using directed *hypergraphs*, where multi-input and -output ML tasks can be expressed as *hyperedges*, and artifacts as nodes.

Hyppo operates as a middleware between the application user and the ML execution engine. Given an input pipeline, it generates an optimized pipeline exploiting optimization opportunities registered by past pipeline executions into a Hyppo structure named *History*. In contrast to prior work, Hyppo considers possible execution plans that exploit both reuse opportunities and task/artifact equivalences. It identifies part of the history that is relevant in terms of reuse and equivalence with the current pipeline, and leverages it to create an augmented pipeline. The augmentation is also represented as a hypergraph, in which some artifacts might have multiple incoming hyperedges, representing alternative ways to obtain them: (a) from storage if materialized, and (b) from equivalent tasks (or sequences thereof). The state of the art on reuse-materialization [1, 8] only exploits the former optimization opportunities.

Architecture. Hyppo's system architecture is illustrated in Figure 1. Its core components include: (a) catalog, (b) parser, (c) augmenter, (d) plan generator, (e) monitor, (f) cost estimator, and (g) history manager. The pipeline optimization process starts with the user submitting their ML code. The parser uses the catalog's dictionary to create a *pipeline* representation P from the raw code. The augmenter then retrieves relevant historical information from the *history* H, to derive the *augmentation* A, which verifies equivalences and encodes alternative options to generate artifacts in the original pipeline. The plan generator retrieves estimations for the cost of tasks and searches the space of alternatives included in the augmentation to derive the optimal execution *plan* p.

[1] https://www.kaggle.com/competitions/higgs-boson/overview

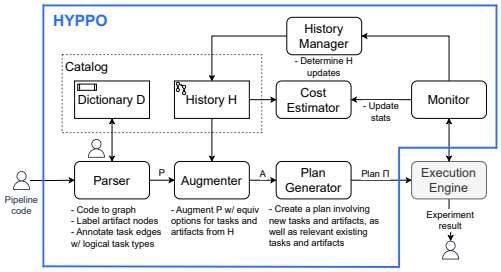

Figure 1: Hyppo's architecture

Dictionary. The task *dictionary* contains an extensible set of equivalent operators and tasks typically met in ML pipelines. The dictionary entries follow the form: $D = \{\ldots, lop_i.tasktype_j : [\ldots, impl_k, \ldots], \ldots\}$ where a task type of a logical operator *lop* is associated with equivalent physical implementations *impl*; the dictionary entries capture also operator configurations—e.g., `Ridge(alpha = 75.0)`—but for ease of presentation we omit a detailed description of the matter. The Hyppo API allows the exploration of the dictionary. For example, in Line 6 of Snippet 1, the data scientist requests to see all entries in the dictionary that relate to classifiers.

History. Hyppo's history is a labelled hypergraph H that compiles the knowledge acquired by previous pipeline executions. It keeps track of all artifacts and tasks, along with their metadata and execution statistics, used by running pipelines. Nodes and hyperedges represent artifacts and tasks, respectively. A labelled node contains metadata such as artifact name, cost, type, size, access frequency, and version. A labelled edge comprises metadata such as task name, cost, type, and the operator id it is associated with. The history records all observed artifacts and tasks that created those artifacts, along with their lineage and useful information such as their cost and whether they are materialized.

The Hyppo API enables the data scientist to programmatically and visually explore the history. Programmatically, in Snippet 1, the data scientist can retrieve performance metrics subject to constraints (Line 3), filter the history down to a single pipeline (Line 4), and retrieve usage statistics subject to constrains (Line 5). Moreover, the data scientist can interact with a visual representation of the graph, and perform filtering and retrieving operations (demonstrated in the video). A visualization of an example history graph is shown in Figure 2 (left).

Cold Starting History. Hyppo's discovery and optimization capabilities depend on the quality of the recorded History. To cold start History, we have created a pipeline generator that uses the Dictionary to construct alternative pipelines for different problems (e.g., classification, regression) that can be used to populate Hyppo's History. Our pipeline generator uses popular operators identified in two studies analyzing research papers from various domains [7] and millions of GitHub repositories and enterprise ML pipelines [5]. Similarly, when a new dataset is introduced to Hyppo, there are no available statistics. To address this, users can use the provided pipeline generator or discover top-performing pipelines for other datasets and apply them to the new dataset. State-of-the-art research [6] has observed that pipelines applied to similar datasets typically produce similar outcomes. This strategy is effective in practical applications despite lacking theoretical assurances.

Parser. The parser processes the input pipeline code and converts it to the *pipeline* labelled hypergraph P. In doing so, it assigns names to artifacts that encode equivalences that can be later exploited. The Hyppo API allows the data scientist to specify a pipeline. For example, Line 7 of Snippet 1 invokes the parser of Hyppo. A visualization of an example input pipeline is shown in Figure 2 (top).

Pipeline Augmenter. The function of the augmenter is to enrich the pipeline P with alternative, equivalent options for task and artifact computation that have been recorded in the history H. This enables a larger space of potential computation and execution options, allowing the Hyppo optimizer to pick a presumably beneficial plan. The outcome of this process is an augmented pipeline, or simply an augmentation A that is a directed hypergraph with the property that the pipeline P is a subhypergraph of A. The augmenter is only indirectly used in the scenario of Section 2, as it is called internally in the call to the plan generator (Line 9 of Snippet 1). An example augmented graph is shown in Figure 2 (right-middle).

Plan Generator. The plan generator seeks to identify an "optimal" plan among those encoded in the augmented pipeline A. Specifically, the plan should optimize for the total execution cost. This is done by reusing materialized artifacts, as well as exploiting equivalent alternative tasks to derive the artifacts contained in the original pipeline. A plan may avoid executing some of the new tasks, when they can be exchanged by cheaper equivalent ones, or sequences thereof. The plan generator is available through the `optimal_plan` API call, as shown in Line 9 of Snippet 1.

Figure 2 depicts the pipeline optimization process of Hyppo. Initially, the data scientist submits a pipeline, termed `pipe` in Snippet 1, shown as P in Figure 2 (top). Hyppo invokes the augmenter that retrieves relevant information from the history to generate the augmentation, shown as A in Figure 2 (right-middle). And finally, Hyppo produces an optimal plan (Line 9 of Snippet 1) shown as Π in Figure 2 (bottom).

Monitor. Hyppo's monitor serves two functions. First, it collects traces of metrics from pipeline execution, e.g., resource utilization, execution time. These are used by the cost estimator to update the statistics maintained for the operators in the dictionary. Second, it monitors the execution of new tasks and the cost of producing the resulting artifacts. This information is used by the graph manager.

Cost Estimator. The cost estimator is responsible for (a) implementing our cost model and (b) updating the statistics in the dictionary `D`. The plan generator probes the cost estimator for computing the costs of tasks involved in the plan according to the cost model. The cost estimator is invoked internally by the plan generator.

History Manager. The history manager (a) keeps track of the execution of new pipelines, maintains the history H accordingly, and (b) decides what artifacts to materialize. The first function involves updating H with new tasks that have been executed and new artifacts that have been generated. As new artifacts are being produced, it also performs the second function, which involves a critical decision: given a storage budget, which artifacts (from both, those already materialized and the newly created with the execution of a plan) to materialize for reducing the computational cost, and thus, the execution time of future pipelines. The outcome of this decision is that certain artifacts that have been materialized will be evicted from storage, while some new artifacts will be stored.

The history manager is internally invoked when the data scientist commits a pipeline for execution, as in Line 10 of Snippet 1.

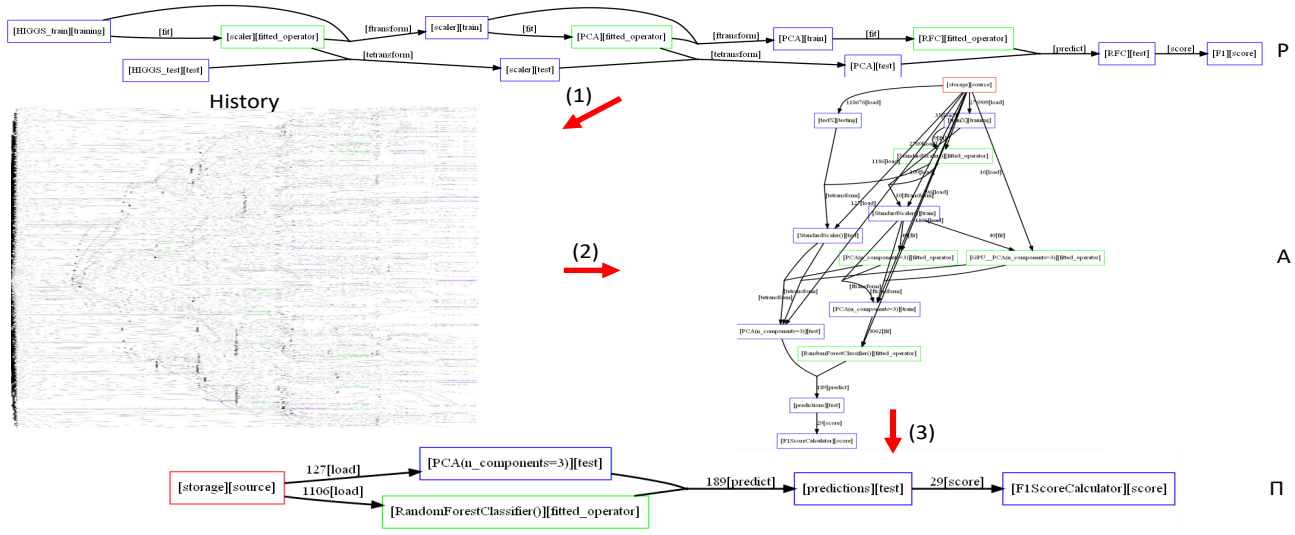

Figure 2: Example history (left) and pipeline graphs (right) of the various steps of Snippet 1

Implementation. Hyppo[2] has been implemented in Python 3.10 and NetworkX (3.1) (parser, optimizer). Hyppo's History is stored in Neo4j (4.4) and its API uses the graph query language Cypher. Our catalog supports 40+ popular operators from popular ML libraries, and it keeps growing. Our experiments reveal that Hyppo improves artifact and model retrieval time by an order of magnitude and total optimization time by 5.1x compared to prior state of the art [1, 8], and it is scalable.

4 OUR PRESENTATION

We will introduce the audience to Hyppo's (a) discovery and optimization features, (b) extensibility, and (c) the system's internals. In the main interactive scenario, the audience is provided with a selection of ten diverse datasets, each accompanied by distinct use cases. This variety ensures that every participant can explore and interact with a dataset that aligns with their interests or background.

Scenario. Continuing the main demonstration scenario (Section 2), we will further demonstrate pipeline optimization. Starting with the discovered pipelines, we will investigate different optimization strategies: (a) no optimization, (b) computation sharing, cost-based reuse, and materialization, but without history [8], (c) computation sharing, heuristic-based reuse and materialization [1], and (d) Hyppo's approach, with computation sharing, materialization, and equivalence optimization strategies. After the main demonstration, we will have additional end-to-end canned scenarios that further showcase Hyppo's features.

Extensibility. Hyppo is designed for easy extension, supporting the addition of both new logical and physical operators. This capability will be showcased through audience participation, where:

(1) Participants can introduce new logical operators not currently found in the Dictionary via the notebook. (2) They can create and assess pipelines utilizing these new operators. (3) Likewise, new implementations of existing logical operators can be introduced and evaluated. This process will reveal how Hyppo can uncover previously undiscovered pipelines and leverage equivalent physical operators to generate better execution plans, especially if the new implementations enhance performance.

System's Internals. For off-script presentation and discussion, we will provide further interactivity where (1) the participants can browse example pipelines and experiments or even create their own pipelines leveraging our operator dictionary and a pre-populated pipeline history. (2) Participants interested in our graph database can review the graph database schema, nodes, and relationship types and their respective properties. Lastly, (3) participants interested in optimization can use Hyppo to generate, estimate the cost, and visualize numerous execution plans given a pipeline, allowing us to engage in insightful discussions.

Acknowledgments: This work has been partially supported by the H2020-MSCA-ITN-2020 DEDS (GA.955895), the European Union's Horizon Europe Research and Innovation programmes DataGEMS (GA.101188416), CREXDATA (GA.101092749), ExtremeXP (GA.101093164), and the Spanish Ministerio de Ciencia e Innovación under project PID2020-117191RB-I00/AEI/10.13039/50 1100011033 (DOGO4ML).

[2]Hyppo code repo: https://github.com/akontaxakis/HYPPO

REFERENCES

[1] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*.
[2] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowl. Based Syst.* 212 (2021), 106622.
[3] Antonios Kontaxakis, Dimitris Sacharidis, Alkis Simitsis, Alberto Abelló, and Sergi Nadal. 2024. HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning. In *ICDE*.
[4] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-Grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*.
[5] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (2022).
[6] Joaquin Vanschoren. 2019. *Meta-Learning*. Springer International Publishing, Cham, 35–61. https://doi.org/10.1007/978-3-030-05318-5_2
[7] Doris Xin, Litian Ma, Shuchen Song, and Aditya G. Parameswaran. 2018. How Developers Iterate on Machine Learning Workflows - A Survey of the Applied Machine Learning Literature. *CoRR* abs/1803.10311 (2018).
[8] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018).
[9] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization Optimizations for Feature Selection Workloads. *ACM Trans. Database Syst.* 41, 1 (2016).