**ABSTRACT**

Data application developers and data scientists spend an inordinate amount of time iterating on machine learning (ML) workflows—by modifying the data pre-processing, model training, and post-processing steps—via trial-and-error to achieve the desired model performance. Existing work on accelerating machine learning focuses on speeding up one-shot execution of workflows, failing to address the incremental and dynamic nature of typical ML development. We propose Helix, a declarative machine learning system that accelerates iterative development by optimizing workflow execution end-to-end and across iterations. Helix minimizes the runtime per iteration via program analysis and intelligent reuse of previous results, which are selectively materialized—trading off the cost of materialization for potential future benefits—to speed up future iterations. Additionally, Helix offers a graphical interface to visualize workflow DAGs and compare versions to facilitate iterative development. Through two ML applications, in classification and in structured prediction, attendees will experience the succinctness of Helix’s programming interface and the speed and ease of iterative development using Helix. In our evaluations, Helix achieved up to an order of magnitude reduction in cumulative run time compared to state-of-the-art machine learning tools.

**PVLDB Reference Format:**


DOI: https://doi.org/10.14778/3229863.3236234

1. INTRODUCTION

Development of real-world machine learning applications typically begins with a simple workflow, which evolves over time as application developers iterate on it to improve performance. Using existing tools, every single small change to the workflow results in several hours of recomputation from scratch, even though the change may only affect a small portion of the workflow. For example, changing the regularization parameter should only retrain the model but not rerun data pre-processing. One approach to mitigate this expensive recomputation is to materialize every single intermediate that does not change across iterations, but this approach requires programming overhead to keep track of changes across iterations, as well as to deal with how and when to materialize intermediates, and to reuse them in subsequent iterations. Since this is so cumbersome, developers often opt to instead rerun the entire workflow from scratch.

Unfortunately, existing machine learning systems fail to provide robust support for rapid iteration on machine learning workflows. For example, KeystoneML [1] aims at optimizing the one-shot execution of workflows by applying techniques such as common subexpression elimination and caching. Columbus [15] focuses on optimizing multiple feature selection steps within one iteration. DeepDive [14], targeted at knowledge-base construction, materializes the results of all feature extraction and engineering steps. While this naïve materialization approach speeds up iterative development in certain settings, it can be wasteful and time-consuming.

We demonstrate Helix, a declarative, general-purpose end-to-end machine learning system that accelerates iterative machine learning application development with three key features:

**Declarative domain specific language.** Data scientists write code in a simple, intuitive, and modular domain-specific language (DSL) built on Scala, while also employing UDFs as needed for imperative code, say for feature extraction or transformation. This inter-operability allows data scientists to leverage existing functions and libraries on JVM and Spark-specific operators.

**Iterative execution optimization.** Helix represents the machine learning workflow programmed in our DSL as a directed acyclic graph (DAG) of data collections. For each node (representing an intermediate result), Helix decides whether to materialize it by considering the maximum storage budget, the time to compute the node and all of its ancestors, and the size of the output—this is the materialization problem. Then, during subsequent iterations, Helix determines whether to read the result for a node from persistent storage (if previously materialized), or to compute it from the input—this is the recomputation problem. We found that recomputation is in PTIME via a non-trivial reduction to MAX-FLOW using the PROJECT SELECTION PROBLEM [3], while materialization is NP-HARD via a reduction from the KNAPSACK PROBLEM. We propose a simple cost model used in an online algorithm to provide an approximate solution to the materialization problem. Figure 2 (described later) shows that Helix provides 60% to an order of magnitude reduction in cumulative run time reduction compared to state-of-the-art tools like DeepDive and KeystoneML.

**Workflow versioning and visualization tool.** We build a versioning and visualization tool on top of Helix, enabling the management of workflow versions, execution plan visualization, and version comparison. Users can easily track the evolution of a workflow, including the changes to hyperparameters, feature selection, and the performance impact of each modification on the workflow. Our demonstration aims to: a) Highlight Helix’s succinct yet flexible declarative DSL for programming end-to-end machine learning workflows. b) Demonstrate how Helix accelerates iterative machine learning application development by providing (1) End-to-end optimization of the entire workflow; (2) Automatic detection of the operator changes; (3) Intelligent materialization of inter-
mediate results for maximizing reuse in subsequent executions. c) Show how HELIX’s graphical interface can support debugging and result analysis during workflow development. The attendees will be able to interact with the HELIX system via a graphical interface that includes four main modules: code editor, workflow DAG visualization tool (shown side-by-side with the code), workflow versions browser, and workflow version comparison tool. The DAG visualization tool helps users explore optimizations to the execution plan. The version browser and comparison tool allows users to gain insights into relationships among features, models, and performance metrics, thus providing developers effective guides on how to fine-tune the model to save exploration time.

Note that the techniques and abstractions involved in building HELIX are general—wrappers for other ML and data processing frameworks can be easily implemented while using the same core optimization engines and programming model.

2. SYSTEM OVERVIEW

The HELIX backend comprises a domain-specific language (DSL) in Scala as the programming interface, a compiler for the DSL, and an execution engine. Figure 1(a) illustrates the backend architecture. The compiler first translates the program written in the DSL into a DAG of intermediate results (associated with the corresponding operators that generated them), which is then optimized to minimize overall execution time, by pruning extraneous operations (or equivalently, intermediate results), reordering operations, and reusing results from previous iterations when applicable. The execution engine uses an online algorithm that determines at runtime the set of intermediate results to materialize in order to minimize execution time for subsequent iterations. We provide a brief overview of each of these three components below.

2.1 Programming Interface

HELIX’s DSL is akin to KeystoneML’s DSL for constructing ML pipelines, with the added benefits of user-friendly data structures for data pre-processing. HELIX users program their entire procedure in a single Scala interface called Workflow. Users can directly embed Scala expressions as user-defined functions (UDFs) into declarative statements in the DSL, in the same fashion that SparkSQL supports inline SQL UDF registration [1]. Figure 1(a) shows an example workflow in the HELIX DSL for the Census application that will be described in Section 3. The DSL facilitates elaborate data pre-processing and complex machine learning (ML) model development with the following features. With a handful of operator types, the DSL supports both fine-grained and coarse-grained feature engineering, as well as both supervised and unsupervised learning. The DSL has been used to implement workflows in social sciences, information extraction, computer vision, and natural sciences, all under 100 lines of code per workflow. Users can easily extend the default set of operators to adapt to their custom use cases by providing only the UDF without copying boilerplate code. HELIX’s data structure for pre-processing maintains features in human-readable format for ease of development and automatically converts it into a compatible format for ML.

2.2 Compiler

During the compilation phase, high-level DSL declarations in a Workflow are first translated into a DAG of operations (or equivalently, intermediate results) using the intermediate code generator. Figure 1(b) shows an example of the operations DAG compiled from the program in Figure 1(a). The DAG optimizer analyzes the generated DAG along with relevant data, including the input data and any materialization of intermediate results from previous executions, to produce a physical execution plan, with the optimization objective of minimizing the latency of the current iteration. This involves several components:

Iterative change tracker. To avoid the inefficiencies of rerunning invariant operations, HELIX automatically detects changes to an operator from the last iteration and invalidates all results affected by the changes via dependency analysis. Unfortunately, the problem of determining operator equivalence for arbitrary functions is undecidable as per Rice’s Theorem [9], with extensive bodies of work in the programming language community dedicated to solving it for specific classes of programs. Currently, HELIX supports change detection via source code version control; covering more general cases is future work. Figure 1(b) shows highlighted changes automatically detected by HELIX between two versions of a workflow (+/- indicates statements that are added/deleted).

Program slicing component. HELIX applies program slicing techniques from compilers to prune extraneous operations that do not contribute to the final results. Feature selection is a prevalent practice in machine learning, and this component uses fine-grained data provenance to automatically eliminate computation for features that do not impact the model, without any code change by the user.

Recomputation component. The DAG optimizer in the compiler determines the optimal reuse policies that minimize execution time of the current iteration given results from previous iterations. Formally, let \( G = (N, E) \) be a DAG of operations. Each node \( n_i \in N \) has a compute cost \( c_i \) and a load cost \( l_i \). Additionally, each node is assigned a state from \( S = \{ load, compute, prune \} \), with the prune constraint that stipulates that a node in compute cannot have parents in prune (i.e., the parents of a node must be available for that node to be computed). Let \( s : N \to S \) be the state assignments and \( I \) be the indicator function. The objective of the recomputation problem is finding \( s^* \):

\[
\argmin_s \sum_{n_i \in N} I(s(n_i) = compute)c_i + I(s(n_i) = load)l_i \quad (1)
\]

This cannot be solved via a simple traversal of the DAG due to the prune constraint. While loading a node \( n_i \) allows us to prune all of its ancestors \( A(n_i) \), the actual run time reduction incurred by loading \( n_i \) depends on the states of all descendants of each \( n_j \in A(n_i) \). For example, if \( l_i > c_k \) for a node \( n_k \) that is a child of some \( n_j \in A(n_i) \), the run time is minimized by keeping \( n_j \) and computing \( n_k \) from it. We prove that this problem is polynomial-time reducible to the \textsc{Project Selection Problem} [12], a variant of \textsc{Max-Flow}, and devise an efficient \textsc{PTIME} algorithm to compute the optimal plan via this reduction [12].

Figure 1(b) shows an example optimized plan. Each node corresponds to the result of an operator declared in Figure 1(a), with operators for data pre-processing in purple and machine learning in orange. Nodes with a drum to the left are reloaded from disk, whereas nodes with a drum to the right are materialized. Operators in the source code that are pruned during execution are grayed out. Iterative changes to the code are highlighted in red and green in 1(b).

2.3 Execution Engine

The execution engine executes the physical plan produced by the compiler, using Spark [13] as the main backend for data processing, supplemented with JVM libraries for application-specific needs.

During execution, the \textsc{materialization optimizer} chooses intermediate results to persist (with a maximum storage constraint) in order to minimize the latency of future iterations, using runtime statistics from the current and prior executions for guidance. This optimization problem is complicated by two practical challenges: 1) the total number of iterations the user will perform is not known
1. object Census extends Workflow
2. data is read into rows using CSVScanner
3. age refers to FieldExtractor("age")
4. edu refers to FieldExtractor("education")
5. acc refers to FieldExtractor("income")
6. target refers to FieldExtractor("predictions")
7. ageBucket refers to FieldExtractor("ageBucket")

Figure 1: a) Example workflow in HELIX DSL for the Census application, with +/− indicating iterative changes. b) Optimized execution plan for the modified workflow in a). Operators for data pre-processing are in purple, and machine learning in orange; operators from a) pruned at execution time are grayed out. Nodes highlighted in red and green correspond to the code changes in a). c) System architecture.

3. DEMONSTRATION DESCRIPTION

We will demonstrate the ease and speed of iterative machine learning development using HELIX through two distinct ML applications. We compare with the unoptimized version of HELIX to help attendees appreciate the gains of HELIX’s optimizations.

Applications. 1) Census: This application illustrates a simple classification task with straightforward features from structured input. The dataset from [5] contains demographic information, such as age, education, occupation, used to predict whether a person’s annual income is >50K. The complexity of this application is representative of applications from the social and natural sciences, where well-defined variables are being studied for covariate analysis. Code for this workflow is shown in Figure 1(b). 2) Information Extraction: This is a complex structured prediction task that identifies person mentions from news articles. In contrast to Census, the input to this workflow is unstructured text, and the objective is to extract structured information instead of simple classification. Thus, this workflow requires more data pre-processing steps to enable learning, mirroring the typical industry setting where extensive data ETL is necessary.

3.1 User Interface

Attendees will interact with the HELIX system through a single web application with an IDE for programming and modules for examining results and system details.

IDE. The HELIX IDE provides HELIX DSL specific autocomplete and syntax highlighting to facilitate programming. A “Suggest Modifications” button lets user request machine-generated edits to be shown inline using Github-style code change highlighting, as illustrated in Figure 1(a), thus allowing users to iterate rapidly on the workflow without mastering the DSL. Once the workflow is executed, the user will be able to inspect the optimized execution plan in the DAG format, as shown in Figure 1(b). Individual runtime and storage for each operation are displayed by hovering over them.
4. RELATED WORK

A great deal of work focuses on development of end-to-end systems for common ML operations, focusing on expressiveness for ML tasks at the language level [14] or API level [2, 6] and provides first-class support for tasks such as model selection [10], workflow construction [8], feature selection [15], and feature engineering [4]. Such systems typically optimize the runtime of ML pipelines on a single node [2] or in a distributed setting [11, 1, 4]. Another common theme is the specification of machine learning tasks through an expressive and easy-to-use declarative programming model [14]. HELIX shares some characteristics with these systems in that it adopts many of the same goals as secondary considerations, but is unique in that it identifies iterative development as a primary concern and is the first system to implement novel, principled solutions for this particular focus.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grants IIS-1513407, IIS-1633755, IIS-1652750, and IIS-1733878 awarded by the National Science Foundation, and funds from Microsoft, 3M, Adobe, Toyota Research Institute, Google, and the Siebel Energy Institute. Doris Xin and Stephen Macke were supported by National Science Foundation Graduate Research Fellowship grants NSF DGE-1144245 and NSF DGE-1746047. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

5. REFERENCES