# HELIX: Accelerating Human-in-the-loop Machine Learning

Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, Aditya Parameswaran
University of Illinois (UIUC)
{dorx0,litianm2,jialin2,smacke,ssong18,adityagp}@illinois.edu

## 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:**
Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, Aditya Parameswaran. Helix: Accelerating Human-in-the-loop Machine Learning. *PVLDB*, 11 (12): 1958 - 1961, 2018.
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 [11] 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 interoperability 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-


*Proceedings of the VLDB Endowment,* Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236234

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 1c) 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 1a) 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 1b) shows an example of the operations DAG compiled from the program in Figure 1a). 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 1a) 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 $n_i \in N$ 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 $s : N \rightarrow S$ be the state assignments and $\mathbb{I}$ be the indicator function. The objective of the recomputation problem is finding $s$:

$$\underset{s}{\operatorname{argmin}} \sum_{n_i \in N} \mathbb{I}\{s(n_i) = compute\}c_i + \mathbb{I}\{s(n_i) = load\}l_i \quad (1)$$

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 $l_k \gg c_k$ 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 PROJECT SELECTION PROBLEM [3], a variant of MAX-FLOW, and devise an efficient PTIME algorithm to compute the optimal plan via this reduction[12].

Figure 1b) shows an example optimized plan. Each node corresponds to the result of an operator declared in Figure 1a), 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 1a).

## 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 *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

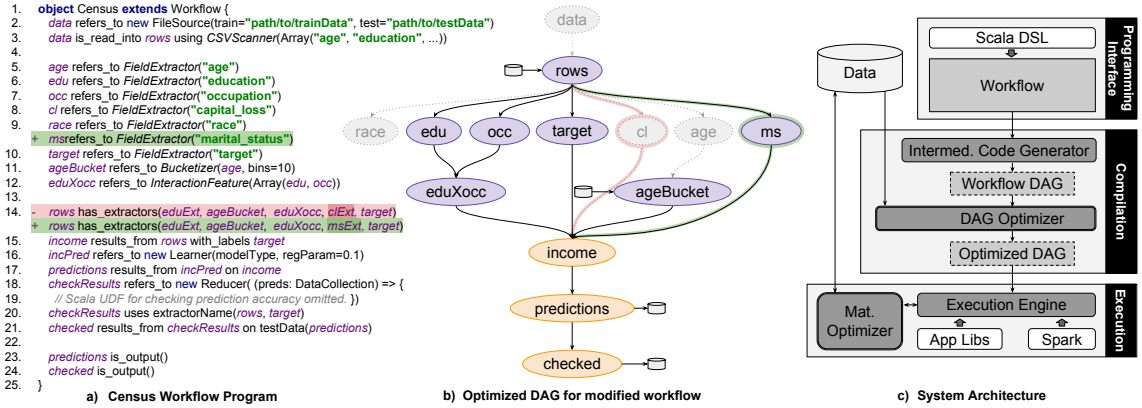


**Figure 1:** a) Example workflow in HELIX DSL for the Census application, with +/− indicating iterative changes. b) Optimized execution plan for the modified workflow in a). Operators for data pre-processing are in purple, and machine learning in orange; operators from a) pruned at execution time are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

a-priori; 2) changes to the workflow in future iterations are unpredictable, making it difficult to determine the intermediate results that can be reused in iterations that follow. Even in the simplest case, with the strong assumption that the user will carry out only one more iteration, and all results from the current iteration will be reusable in the next, it can be shown, via a reduction from the KNAPSACK PROBLEM, that this optimization problem is still NP-HARD. Additionally, the decision to materialize must take place immediately upon operation completion, as it is prohibitive to cache multiple intermediate results for deferred decisions. Thus, an online algorithm is needed to make decisions quickly in real-time. We use a simple cost model to determine the set of intermediate results to materialize as they become available. Recall an operator $n_i$ is associated with a load cost $l_i$ and a compute cost $c_i$. At iteration $t$, the reduction in execution time at iteration $t+1$, from materializing $n_i$ at $t$, can be approximated as $r_i = 2l_i - (c_i + \sum_{n_j \in A(n_i)} c_j)$. If $r_i$ is negative and the data size for $n_i$ is less than the remaining storage budget, then materialize $n_i$. Although this model ignores the dependencies between other operators and $A(n_i)$, it is cheap to compute and effective in practice, while satisfying the online constraint. Our ongoing work investigates predicting reuse probability based on user studies and workflow features.

## 2.4 Performance Gains

We show preliminary experiments comparing HELIX with two similar ML systems, DeepDive [14] and KeystoneML [11], on the two applications to be described in Section 3. KeystoneML is absent in Figure 2(a) because it is not equipped to handle information extraction (IE) tasks, whereas DeepDive has missing data for iteration $> 2$ in Figure 2(b) because its ML and evaluation components are not user-configurable. To show the type of modification in each iteration, we use purple to indicate a data pre-processing change (e.g., adding a feature), orange for ML (e.g., adding regularization), and green for evaluation (e.g., changing metrics).

Figure 2(a) shows that for the IE task, HELIX's cumulative run time is *60% lower than that of DeepDive*, due to judicious materialization of only intermediates that help with future iterations, in contrast to DeepDive's materialize-all approach. Figure 2(b) shows all three systems' performance on a classification task, where HELIX shows nearly *an order of magnitude reduction* in cumulative run time. Note that in post processing iterations (green), HELIX has near zero runtimes due to high reuse rate. ML iterations (orange) has slightly higher runtime but less than data pre-processing iterations (purple), which have the least amount of reuse. For a never-materialize system such as KeystoneML, the rerun time is constantly large regardless of what has been changed.

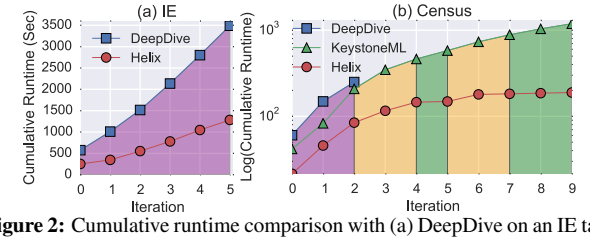


**Figure 2:** Cumulative runtime comparison with (a) DeepDive on an IE task. (b) DeepDive and KeystoneML on a classification task.

# 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 1a). *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 1a), 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 1b). Individual runtime and storage for each operation are displayed by hovering over them.

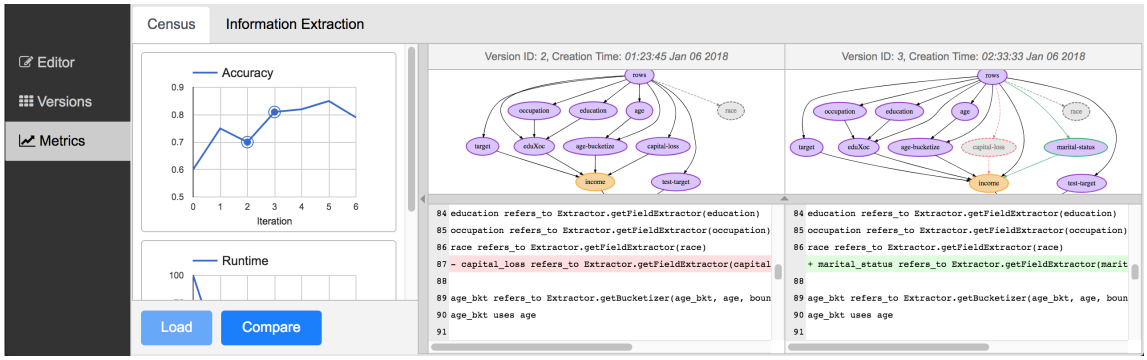


**Figure 3:** Metrics aggregation and version comparison. Users can select and compare specific versions, represented as points on the metric trend lines, for code change and visualized execution plans, in order to better understand the performance impact of specific modifications.

**Versions.** Users can quickly browse through all past versions of a workflow in a summarized view with similar aesthetics to code version control tools such as git. Each version is shown as a commit log entry, with buttons that allow users to instantly *checkout* the code or obtain additional metadata. We also provide shortcuts to the version with the best evaluation metrics as well as the latest version at the top of the page.

**Metrics.** As shown in Figure 3, the Metrics tab aggregates the evaluation metrics for the workflow across iterations into plots with the metric value on the y-axis and the iteration number on the x-axis. Each point in the plot represents a version of the workflow. Users can select a single point to load the associated code version or two points for comparison. In Figure 3, Version 2 and 3 are selected in the Accuracy plot for comparison. The comparative view visualizes the DAG and highlights changes in the DAG using git-like visual comparison cues, in addition to showing the two versions of the workflow code also with changes highlighted. This feature enables rapid exploration of the relationships between various metrics and changes to specific components of the workflow. Understanding the impact of each past iteration is crucial for making effective future improvements, thus reducing the overall number of iterations to achieve the desired outcome.

## 3.2 Guided Interaction

Once acquainted with the system and applications, attendees are invited to select an application and execute its initial version. Upon completion, we will describe each component of the UI using the initial results. Attendees are then invited to modify the workflow in the IDE to optimize for either the prediction accuracy or overall runtime to achieve a certain level of accuracy. They can also use machine-generated suggested edits to the workflow for quick exploration without learning the DSL. The version browser can be used to see past changes or roll back to a past version and branch out in another direction. After the first modified version is executed, we will compare the execution plan with the one from the previous iteration to showcase the workflow change detection feature, which allows HELIX to automatically reuse results for operators not affected by the changes.

Attendees can investigate the relationship between past versions and metric values using the Metrics tab to inform decisions on what to try next. To emphasize the benefits of HELIX's optimizations, we will execute the same version twice, once with and once without optimizations, and compare the runtimes and execution plans. At the end of the session, attendees are invited to review a summary of their interactions via the Versions and Metrics tabs to gain a better appreciation for the progress they made in a short period of time.

# 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, 4] or API level [7, 6] and provides first-class support for tasks such as model selection [10], workflow construction [8], feature selection [15], and feature engineering [14]. 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, 4]. 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

[1] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
[2] X. Feng et al. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.
[3] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education, 2006.
[4] T. Kraska et al. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.
[5] M. Lichman. UCI machine learning repository, 2013.
[6] C. D. Manning et al. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, 2014.
[7] X. Meng et al. Mllib: Machine learning in apache spark. *JMLR*, 2016.
[8] F. Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR*, 2011.
[9] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 1953.
[10] E. R. Sparks et al. Tupaq: An efficient planner for large-scale predictive analytic queries. *arXiv preprint arXiv:1502.00068*, 2015.
[11] E. R. Sparks et al. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, 2017.
[12] D. Xin et al. Helix: Holistic optimization for accelerating iterative machine learning. *Technical Report http://data-people.cs.illinois.edu/helix-tr.pdf*, 2018.
[13] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[14] C. Zhang. *DeepDive: a data management system for automatic knowledge base construction*. PhD thesis, The University of Wisconsin-Madison, 2015.
[15] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.*, 2016.