Accelerating Human-in-the-loop Machine Learning: Challenges and Opportunities

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

ABSTRACT

Development of machine learning (ML) workflows is a tedious process of *iterative experimentation*: developers repeatedly make changes to workflows until the desired accuracy is attained. We describe our vision for a "human-in-the-loop" ML system that accelerates this process: by intelligently tracking changes and intermediate results over time, such a system can enable rapid iteration, quick responsive feedback, introspection and debugging, and background execution and automation. We finally describe Helix, our preliminary attempt at such a system that has already led to speedups of upto 10× on typical iterative workflows against competing systems.

1 INTRODUCTION

Due to the unpredictable nature of machine learning (ML) model performance, developing ML applications relies on numerous iterations of trial-and-error—a *step-by-step process of experimentation*[1]. The development process often begins with an initial workflow containing simple data pre-processing and modeling steps. Then, based on analysis of the resulting model, the developer modifies the workflow to improve performance. Examples of such modifications include adding/removing features, introducing new data sources, switching from logistic regression to deep neural nets, adding regularization to the model, and changing the evaluation metrics. Many such iterations take place between the conception and the deployment of the ML model, with the developer as an integral component. We model this process in Figure 1, with the dotted box representing one instance of the ML workflow, with the developer "in-the-loop" using the end results as cues for modifications.

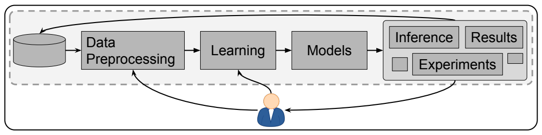

Figure 1: Development Cycle.

Unfortunately, most work on ML systems has focused on specification and acceleration of the one instance of a given ML workflow (i.e., the dotted box in Figure 1), without reasoning about the iterative "human-in-the-loop" aspect. By doing so, such systems have a number of deficiencies:

- ***Iterative reuse:*** *Developers wait for minutes to hours even on small changes to the workflow.* On iterative changes (e.g., changing a feature or regularization), developers rerun workflows from scratch, since it is too cumbersome to identify and reuse intermediate results.
- ***Introspection between iterations:*** *Developers are not able to explain, debug, or understand performance.* Developers do not understand the impact of changes they made to accuracy, and are not able to debug performance across the ML workflow.
- ***Leveraging think-time between iterations:*** *ML systems do not utilize think-time for background processing.* Traditional ML systems do not take advantage of coding or thinking time for background processing, such as trying out workflow alternatives (e.g., changing regularization, training, or doing automated feature selection).
- ***Reducing iterative feedback latency:*** *ML systems do not optimize for quick feedback to developers.* Traditional ML systems do not provide rapid, approximate feedback that allows developers to make decisions before the computation is carried to full completion.
- ***Automated cues for iterative changes:*** *ML systems do not provide novice developers cues for subsequent changes.* Traditional ML systems do not provide suggestions for next steps in the iterative evolution of the workflow.

In this paper, we describe our vision for an *accelerated human-in-the-loop machine learning (HILML) system* that is targeted at eliminating the deficiencies identified above. The ultimate goal of a HILML system is to *shorten the time to obtain deployable models from scratch.* In addition, since we are focusing on the humans-in-the-loop, we also aim to support declarative specification of ML workflows, targeting novice users of ML, while also providing benefits to expert users.

We also present Helix, our first attempt at a declarative, general-purpose HILML system aimed at accelerating iterative ML application development. We briefly describe the programming interface and system architecture, and demonstrate the effectiveness of Helix with preliminary results on multiple applications.

Related Work. Prior work has recognized the importance of studying many aspects of HILML, such as iterative model building [12], model management both in general [13] and specifically tailored for deep learning [7], dataset versioning [11], and model sharing [6]. We complement existing work with a fresh set of new research challenges. In the general space of ML systems, a number of recent work focus on using declarative programming to improve usability [4, 10, 16]. While some systems aim to support end-to-end ML in a general setting [4, 5, 8, 10], many focus on a special component of the process, such as model selection [9], feature selection [17], and feature engineering [16]. Unlike Helix, these systems are focused on optimization in the single-execution setting, neglecting to consider challenges and opportunities associated with iterative development.

2 OPTIMIZING HUMAN-IN-THE-LOOP ML

In this section, we first describe a concrete, unifying model to ground our discussion and describe prerequisites that any HILML

[1]www.nytimes.com/2014/08/18/technology/ for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

system must obey. Then, in the next subsection, we describe concrete research directions that are enabled by this unifying model.

2.1 Prerequisites: Usability and Model

Usability: Declarativity and Generalization. A basic requirement for HILML systems is that it can generalize across use-cases (spanning applications in the social and natural sciences, for example), workflow design decisions (from supervised to unsupervised learning, and from linear regression to deep neural nets, for example), and expertise levels (from novice to expert). The latter concern is especially important given the demand for ML and AI in a host of new emerging data-driven disciplines. To do so, such a system must accept a declarative or semi-declarative specification (such as recent tools [3, 10, 16]), while also be able to embed arbitrary imperative code.

Workflow DAG Model. Given this declarative specification (with black-box UDFs), a HILML system must be able to capture and abstract a specific instance of a workflow as a *DAG of intermediate data items*, where the nodes in the DAG correspond to the output of specific operators, and the edges indicate input-output relationships between operators. Via the declarative specification, the HILML system can identify the logical operator corresponding to each node in the workflow (e.g., data preparation or model training), giving the system a comprehensive understanding of the workflow.

Between two iterations of the workflow, a specific intermediate data item is deemed to be identical in both iterations if the source code for the corresponding operator has not changed, and recursively the parents of that data item are identical. A source data item is deemed to be identical if the underlying data has not changed between iterations.

To be able to tell what has changed or what has not changed between iterations is not straightforward, however we can detect file-level changes to the source data items, and we can detect code-level changes to the declarative specification; changes to external libraries may be harder to detect, but are likely to not be so frequent.

Our workflow model provides us a valuable starting point for a HILML system—by detecting what is same and what is different across iterations allows a HILML system to understand how the workflow has evolved over iterations.

2.2 HILML Research Challenges

We now describe a number of research challenges that are enabled by our workflow model.

Intermediate Results Reuse. To enable effective reuse of intermediate data items across iterations, we must be able to answer two questions:

1) What intermediates should be materialized in the current execution to speed up future iterations through reuse?
2) What intermediates should be reused to minimize run time given materialized intermediates?

The answers to both questions can be represented as a subset of nodes in a specific instance of the workflow DAG. For 1, notice first that materializing all intermediates may not be beneficial due to high cost that may outweigh any potential benefits. In fact, the answer to 1 is contingent on a number of prediction problems. Since not all intermediates will be reusable given the dynamic nature of the workflow, modeling future savings incurred by reuse requires predicting iterations, both in terms of the number of iterations and the specific modifications. For example, materializing an intermediate data item below a workflow portion that the developer is actively modifying is not likely to be useful, since it may be rendered redundant by the changes to the workflow. Predicting which portion of the workflow a developer may modify next (e.g., maybe the developer is done with feature engineering and have moved onto model tuning) can also help with this decision.

Comparatively, the answer to 2 is more self-contained due to the lack of uncertainty. The complexity for 2 lies within the fact that it needs to take place during compile time, so the system needs to estimate the run time and output size of the operators.

Introspection: analyzing the impact of changes. Knowing statistics such as how certain changes to the workflow have impacted prediction accuracy and overall run time helps developers maximize the utility of future iterations. This goes beyond tracking metrics and data associated with each model version and delves into exploring the causal relationship between performance and specific changes. Recognizing such relationships requires semantic understanding of the workflow, which enables *logical comparison* of the different workflow versions (e.g., version 2 adds a feature and regularization to version 1). The workflow representation provides a means for visualizing the logical difference between two versions of the workflow.

Furthermore, we can help developers identify the set of workflow versions that are the most pertinent to a specific performance gap. This can be framed as a path finding problem in the space of workflow versions, with distance between versions reflecting the amount of logical changes. Specific challenges include distance metric design and characterization of the desired paths.

Automated background search during think time. Based on past usage and semantic understanding of the workflow, the system should be able to automatically identify modifications to the current workflow that lead to potential improvements on the metrics of interest. These changes can be tested in the background while the system is idle (e.g., the developer is writing code or thinking). Note that this ranges beyond grid search for model hyperparameters [4] to include modifications to the data pre-processing components of the workflow as well. The goal is not to cut the human out of the loop, but rather to remove tedious, mechanical iterations from active development time. The developer should spend their time on applying domain knowledge to improve the application instead of exhaustively testing out known tricks.

Quick feedback: end-to-end optimization. Compared to intra-operator optimization (e.g., speeding up model training), end-to-end workflow optimization has a much greater potential for hastening iterations since it is able to capture the higher order, inter-operator inefficiencies missed by intra-operator optimization. Optimizing workflows end-to-end is challenging due to the difficulty of analyzing relationships between operators. Sometimes operators within a workflow can be written in different languages or using different libraries. To enable general end-to-end optimization, we need a common, framework-independent abstraction of operators capable of modeling how data logically flows between operators. Here is a concrete example of end-to-end ML workflow optimization: if a model is sparse (many zero weights), being able to identify the operators corresponding to zero-weight features allows us to prune a large portion of the workflow without compromising accuracy.

Quick feedback: approximate workflow execution. A developer's decision to keep changes made in an iteration depends on the *relative* performance to the previous iteration's. Thus, the results obtained in each iteration do no need to be precise, as long as they accurately indicate the performance trend. The system should provide mechanisms to allow developers to trade *redundant precision* for speed, especially when the data size is large. Approximate computing techniques such as sampling and using low precision floats can be applied to achieved the desired tradeoff between precision loss and speedup. This allows developers to test out the same number of changes in a fraction of the time, with little compromise to their ability to accurately judge the effect of each change.

Automated cues for novice: change recommendations. While expert ML developers may have good intuition how to improve the workflow, inexperienced users can greatly benefit from suggestions on what to try next. Concretely, the system should suggest to the developer what operators to add/delete/modify and the expected outcome. The iteration prediction model discussed above can be adapted for this purpose. Developing such a model requires gathering data on how developers iterate on workflows in various domains, which can be a difficult task since publications tend to focus on results instead of the process. Ideally, the model should consider workflow and data characteristics as well as user skills and system settings. Furthermore, it should learn from the results of previous iterations. Note that we need to be careful not to trap the developer in a local optimum from this feedback loop.

3 HELIX FOR HUMAN-IN-THE-LOOP ML

Helix is our first attempt at a HILML system, satisfying the prerequisites outlined in Section 2.1, and partially addressing the first research challenge in Section 2.2. Helix has a declarative programming interface that is concise yet expressive; it uses the workflow DAG model to enable both end-to-end and cross-iteration optimizations. In this section we provide a brief overview of the system architecture, followed by preliminary results on performance compared to related state-of-the-art systems.

3.1 System Architecture

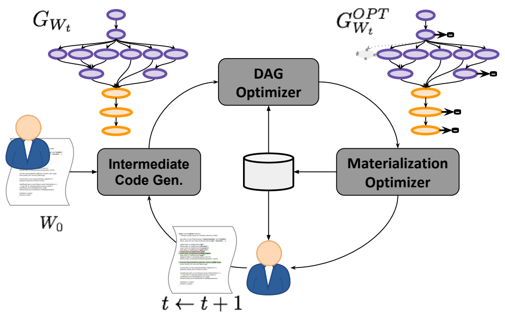

Figure 2: Lifecycle of a Workflow in Helix.

Helix consists of a programming interface, a compiler for the client application code, and an execution engine. The overarching optimization objective is to minimize end-to-end execution time across iterations through intermediate result reuse and redundant operator pruning. As shown in Figure 2, the *materialization optimizer* decides what intermediate results to persist to disk at run time to accelerate subsequent iterations, while the *DAG optimizer* decides what set of intermediates to reload from disk and what operators to prune at compile time to speed up the current iteration. We describe the programming interface and the optimization problems below.

3.1.1 *Programming Interface.*

Helix's programming interface is a DSL embedded in Scala, where the statements are declarative and designed to mimic natural language with infix expressions. A single Scala interface named Workflow is used to program the entire workflow Helix provides a handful of composable and extensible operator types to handle variability and complexity in both data pre-processing and the learning task, ranging from fine-grained to whole-dataset feature engineering, supervised learning to information extraction. Imperative Scala code can be directly embedded into the DSL for user-defined functions (UDFs), similar to inline SQL UDF registration in SparkSQL [2]. Helix uses two distinct data structures to handle data pre-processing and ML, so that data is kept in human-readable formats for easy feature engineering and automatically transformed into ML compatible formats for learning. We have used the DSL to successfully implement workflows in *social sciences, information extraction, computer vision, and natural sciences,* spanning a wide array of data pre-processing and ML use cases. Figure 3 shows a code snippet of the DSL used to program an income prediction task.

3.1.2 *Compilation.*

Helix compiles each Workflow into a DAG of operators. Using both the compiled DAG and relevant data from disk, the DAG optimizer performs three tasks:

Detect changes. Helix automatically detects the set of operators that have changed since the last iteration and marks them for mandatory recomputation. All operators are named, and the DAG optimizer performs light program analysis to compare operators with the same names across iterations for change detection.

Prune redundant operators. Helix prunes extraneous operators by applying dataflow analysis to identify operators that do not contribute to the final output. This feature alleviates the burden of manually removing dead code to avoid redundant computation when data dependencies change.

```
object Census extends Workflow {
  data refers_to new FileSource(train="path/to/trainData", test="path/to/testData")
  data is_read_into rows using CSVScanner(Array("age", "education", ...))

  age refers_to FieldExtractor("age")
  ...
  eduXocc refers_to InteractionFeature(Array(edu, occ))

  rows has_extractors(eduExt, ageBucket,  eduXocc, clExt, target)
  income results_from rows with_labels target
  incPred refers_to new Learner(modelType, regParam=0.1)
  predictions results_from incPred on income
  checkResults refers_to new Reducer( (preds: DataCollection) => {
    // Scala UDF for checking prediction accuracy omitted. })
  checkResults uses extractorName(rows, target)
  checked results_from checkResults on testData(predictions)
  checked is_output()
}
```

Figure 3: Sample code in Helix DSL

Compute the optimal reuse policies. Loading all reusable intermediates from disk is not always the optimal decision for minimizing overall run time. For example, if an operator has a large output but a short compute time, then it is more time-efficient to load its input and recompute. Loading the results of an operator does, however, allow us to prune its inputs, which could have a cascading effect leading to large savings. We can formally model this problem as assigning states to the nodes in the workflow DAG. Each node can be assigned one of three states {compute, load, prune}, and the assignments must satisfy the pruning constraint that a node in the *compute* state must not have parents in the *prune* state. The objective is to find a legal state assignment $s^* =$

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

where $s(n_i)$ is the state of node n_i, and c_i and l_i are the compute and load time, respectively. This problem cannot be solved using a single pass algorithm because of the pruning constraint. We devise an efficient PTIME algorithm to solve Eq (1) optimally by proving that it is polynomial time reducible to MAX-FLOW [14].

3.1.3 EXECUTION ENGINE. HELIX carries out the optimal physical plan produced by the DAG optimizer using Spark [15] for distributed data processing. Note that the core algorithms and optimization techniques in HELIX are independent of the data processing platform. Lightweight wrappers can be written to support other data processing frameworks such as Tensorflow [1].

During execution, the *materialization optimizer*, shown in Figure 2, handles the optimization problem of choosing the intermediates to materialize for reuse in future iterations, under a maximum storage constraint. As discussed in Section 2, the benefit of materializing an intermediate result is dependent on its likelihood of being reused in future iterations. Even with the simplifying assumptions that 1) there will be only one more iteration and 2) all intermediate results are reusable in the next iteration, the problem is still NP-HARD, as we show through a reduction from KNAPSACK [14].

Another complicating factor is that we must make the decision to materialize *online*, i.e., immediately after an operator has completed execution, since deferred decisions are prohibitive as they require caching multiple intermediate results. We propose a simple cost model to achieve an approximate solution while respecting the online constraint. Given the load cost l_i and compute cost c_i for each operator n_i, the cost r_i of materializing n_i is defined as

$$(c_i + \sum_{n_j \in A(n_i)} c_j) - 2l_i \quad (2)$$

We materialize n_i if r_i is negative and l_i is less than the remaining storage. The model naively assumes that loading n_i prunes all of its ancestors from the DAG. While this is untrue as discussed above, we cannot hope to do better given the online constraint. This model has been effective in experimental studies, to be discussed next.

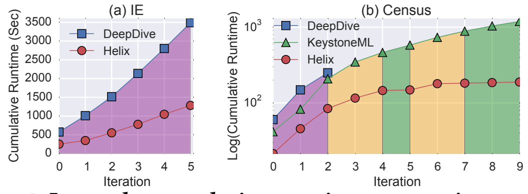

Figure 4: Logscale cumulative runtime comparison with (a) DeepDive on an IE task. (b) DeepDive and KeystoneML on a classification task.

3.2 Performance Gains

We present preliminary results comparing HELIX with two similar ML systems, DeepDive [16] and KeystoneML [10], on an application in information extraction (IE) and another on classification. KeystoneML is not designed to handle IE tasks, hence absent in Figure 4(a). DeepDive has no data for iteration > 2 in Figure 4(b) due to inconfigurable ML and evaluation components. The color under the curve indicates the type of change in each iteration, with purple for data pre-processing, orange for ML, and green for evaluation. The frequencies of each iteration type is determined using statistics collected on 105 applied ML papers [14].

In contrast to DeepDive's materialize-all approach, HELIX judiciously materializes only intermediates that help reduce future run time, resulting in a *60% reduction* in cumulative run time, as shown in Figure 4(a). On the classification task, HELIX achieves *an order of magnitude reduction* in cumulative run time compared to both KeystoneML, which materializes no intermediates, and DeepDive, as shown in Figure 4(b). We see in both workflows that data preprocessing iterations (purple) have the highest iteration run times, while evaluation (green) has the lowest, proportional to the amount of mandatory recomputation for each iteration type.

4 CONCLUSIONS

We presented our vision for an efficient end-to-end ML system focused on supporting iterative, human-in-the-loop workflow development. We identified specific research problems to accelerate and automate workflow development and introduced HELIX—our first attempt at addressing some of these problems.

REFERENCES

[1] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.
[2] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
[3] A. Ghoting et al. Systemml: Declarative machine learning on mapreduce. In *ICDE*, 2011.
[4] T. Kraska et al. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.
[5] X. Meng et al. Mllib: Machine learning in apache spark. *JMLR*, 2016.
[6] H. Miao et al. On model discovery for hosted data science projects. In *DEEM*, 2017.
[7] H. Miao et al. Towards unified data and lifecycle management for deep learning. In *ICDE*, pages 571–582. IEEE, 2017.
[8] F. Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR*, 2011.
[9] E. R. Sparks et al. Tupaq: An efficient planner for large-scale predictive analytic queries. *arXiv preprint arXiv:1502.00068*, 2015.
[10] E. R. Sparks et al. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, 2017.
[11] T. van der Weide et al. Versioning for end-to-end machine learning pipelines. In *DEEM*, 2017.
[12] M. Vartak et al. Supporting fast iteration in model building. In *NIPS Workshop LearningSys*, 2015.
[13] M. Vartak et al. Modeldb: a system for machine learning model management. In *HILDA*, page 14. ACM, 2016.
[14] D. Xin et al. Helix: Holistic optimization for accelerating iterative machine learning. *Technical Report http://data-people.cs.illinois.edu/helix-tr.pdf*, 2018.
[15] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[16] C. Zhang. *DeepDive: a data management system for automatic knowledge base construction.* PhD thesis, The University of Wisconsin-Madison, 2015.
[17] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.*, 2016.