HELIX: Holistic Optimization for Accelerating Iterative Machine Learning

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

ABSTRACT

Machine learning workflow development is a process of trial-and-error: developers *iterate* on workflows by testing out small modifications until the desired accuracy is achieved. Unfortunately, existing machine learning systems focus narrowly on model training—a small fraction of the overall development time—and neglect to address iterative development. We propose HELIX, a machine learning system that *optimizes the execution across iterations*—intelligently caching and reusing, or recomputing intermediates as appropriate. HELIX captures a wide variety of application needs within its Scala DSL, with succinct syntax defining unified processes for data preprocessing, model specification, and learning. We demonstrate that the reuse problem can be cast as a MAX-FLOW problem, while the caching problem is NP-HARD. We develop effective lightweight heuristics for the latter. Empirical evaluation shows that HELIX is not only able to handle a wide variety of use cases in one unified workflow but also much faster, providing run time reductions of up to $\mathbf{19}\times$ over state-of-the-art systems, such as DeepDive or KeystoneML, on four real-world applications in natural language processing, computer vision, social and natural sciences.

PVLDB Reference Format:
Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, Aditya Parameswaran. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB*, 12(4): 446-460, 2018.
DOI: https://doi.org/10.14778/3297753.3297763

1. INTRODUCTION

From emergent applications like precision medicine, voice-controlled devices, and driverless cars, to well-established ones like product recommendations and credit card fraud detection, machine learning continues to be the key driver of innovations that are transforming our everyday lives. At the same time, developing machine learning applications is time-consuming and cumbersome. To this end, a number of efforts attempt to make machine learning more declarative and to speed up the model training process [11].

However, the majority of the development time is in fact spent *iterating on the machine learning workflow* by incrementally modifying steps within, including (i) *preprocessing*: altering data cleaning or extraction, or engineering features; (ii) *model training*: tweaking hyperparameters, or changing the objective or learning algorithm; and (iii) *postprocessing*: evaluating with new data, or generating additional statistics or visualizations. These iterations are necessitated by the difficulties in predicting the performance of a workflow *a priori*, due to both the variability of data and the complexity and unpredictability of machine learning. ***Thus, developers must resort to iterative modifications of the workflow via "trial-and-error" to improve performance.*** A recent survey reports that less than 15% of development time is actually spent on model training [41], with the bulk of the time spent iterating on the machine learning workflow.

Example 1 (Gene Function Prediction)**.** *Consider the following example from our bioinformatics collaborators who form part of a genomics center at the University of Illinois [53]. Their goal is to discover novel relationships between genes and diseases by mining scientific literature. To do so, they process published papers to extract entity—gene and disease—mentions, compute embeddings using an approach like word2vec [40], and finally cluster the embeddings to find related entities. They repeatedly iterate on this workflow to improve the quality of the relationships discovered as assessed by collaborating clinicians. For example, they may (i) expand or shrink the literature corpus, (ii) add in external sources such as gene databases to refine how entities are identified, and (iii) try different NLP libraries for tokenization and entity recognition. They may also (iv) change the algorithm used for computing word embedding vectors, e.g., from word2vec to LINE [60], or (v) tweak the number of clusters to control the granularity of the clustering. Every single change that they make necessitates waiting for the entire workflow to rerun from scratch—often multiple hours on a large server for each single change, even though the change may be quite small.*

As this example illustrates, the key bottleneck in applying machine learning is ***iteration****—every change to the workflow requires hours of recomputation from scratch, even though the change may only impact a small portion of the workflow*. For instance, normalizing a feature, or changing the regularization would not impact the portions of the workflow that do not depend on it—and yet the current approach is to simply rerun from scratch.

One approach to address the expensive recomputation issue is for developers to explicitly materialize all intermediates that do not change across iterations, but this requires writing code to handle materialization and to reuse materialized results by identifying changes between iterations. Even if this were a viable option, materialization of all intermediates is extremely wasteful, and figuring out the optimal reuse of materialized results is not straightforward. Due to the cumbersome and inefficient nature of this approach, developers often opt to rerun the entire workflow from scratch.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297763

Unfortunately, existing machine learning systems do not optimize for rapid iteration. For example, KeystoneML [56], which allows developers to specify workflows at a high-level abstraction, only optimizes the one-shot execution of workflows by applying techniques such as common subexpression elimination and intermediate result caching. On the other extreme, DeepDive [75], targeted at knowledge-base construction, materializes the results of all of the feature extraction and engineering steps, while also applying approximate inference to speed up model training. Although this naïve materialization approach does lead to reuse in iterative executions, it is wasteful and time-consuming.

We present HELIX, a *declarative, general-purpose machine learning system that optimizes across iterations*. HELIX is able to match or exceed the performance of KeystoneML and DeepDive on one-shot execution, while providing ***gains of up to 19***× on iterative execution across four real-world applications. By optimizing across iterations, HELIX allows data scientists to avoid wasting time running the workflow from scratch every time they make a change and instead run their workflows in time proportional to the complexity of the change made. HELIX is able to thereby substantially increase developer productivity while simultaneously lowering resource consumption.

Developing HELIX involves two types of challenges—challenges in *iterative execution optimization* and challenges in *specification and generalization*.

Challenges in Iterative Execution Optimization. A machine learning workflow can be represented as a directed acyclic graph, where each node corresponds to a collection of data—the original data items, such as documents or images, the transformed data items, such as sentences or words, the extracted features, or the final outcomes. This graph, for practical workflows, can be quite large and complex. One simple approach to enable iterative execution optimization (adopted by DeepDive) is to materialize every single node, such that the next time the workflow is run, we can simply check if the result can be reused from the previous iteration, and if so, reuse it. Unfortunately, this approach is not only wasteful in storage but also potentially very time-consuming due to materialization overhead. Moreover, in a subsequent iteration, it may be cheaper to recompute an intermediate result, as opposed to reading it from disk.

A better approach is to determine whether a node is worth materializing by considering both the time taken for computing a node and the time taken for computing its ancestors. Then, during subsequent iterations, we can determine whether to read the result for a node from persistent storage (if materialized), which could lead to large portions of the graph being pruned, or to compute it from scratch. In this paper, we prove that the reuse plan problem is in PTIME via a non-trivial ***reduction to*** MAX-FLOW ***using the*** PROJECT SELECTION PROBLEM ***[28], while the materialization problem is, in fact,*** NP-HARD.

Challenges in Specification and Generalization. To enable iterative execution optimization, we need to support the specification of the end-to-end machine learning workflow in a high-level language. This is challenging because data preprocessing can vary greatly across applications, often requiring ad hoc code involving complex composition of declarative statements and UDFs [7], making it hard to automatically analyze the workflow to apply holistic iterative execution optimization.

We adopt a hybrid approach within HELIX: developers specify their workflow in an ***intuitive, high-level domain-specific language (DSL) in Scala*** (similar to systems like KeystoneML), using ***imperative code as needed for UDFs***, say for feature engineering. This interoperability allows developers to seamlessly integrate existing JVM machine learning libraries [61, 50]. Moreover, HELIX is built on top of Spark, allowing data scientists to leverage Spark's parallel processing capabilities. We have developed a GUI on top of the HELIX DSL to further facilitate development [68].

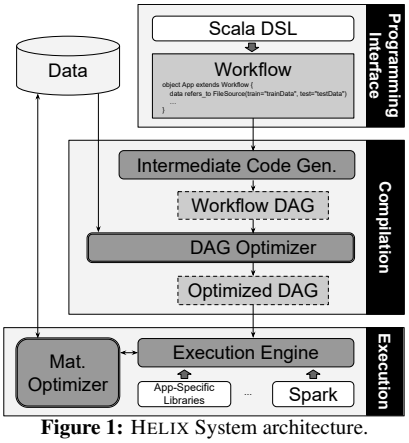

Figure 1: HELIX System architecture.

HELIX's DSL not only enables automatic identification of data dependencies and data flow, but also encapsulates all typical machine learning workflow designs. All of the functions in Scikit-learn's (a popular ML toolkit) can be mapped to functions in the DSL [70], allowing HELIX to easily capture applications ranging from natural language processing, to knowledge extraction, to computer vision. Moreover, by studying the variation in the dataflow graph across iterations, HELIX is able to identify reuse opportunities across iterations. Our work is a first step in a broader agenda to improve human-in-the-loop ML [67].

Contributions and Outline. The rest of the paper is organized as follows: Section 2 presents an architectural overview of the system, and a concrete workflow to illustrate concepts discussed in the subsequent sections; Section 3 describes the programming interface; Section 4 discusses HELIX system internals, including the workflow DAG generation and change tracking between iterations; Section 5 formally presents the two major optimization problems in accelerating iterative ML and HELIX's solution to both problems. We evaluate our framework on four workflows from different applications domains and against two state-of-the-art systems in Section 6. We discuss related work in Section 7.

2. BACKGROUND AND OVERVIEW

In this section, we provide a brief overview of machine learning workflows, describe the HELIX system architecture and present a sample workflow in HELIX that will serve as a running example.

A machine learning (ML) workflow accomplishes a specific ML task, ranging from simple ones like classification or clustering, to complex ones like entity resolution or image captioning. Within HELIX, we decompose ML workflows into three components: data preprocessing (DPR), where raw data is transformed into ML-compatible representations, learning/inference (L/I), where ML models are trained and used to perform inference on new data, and postprocessing (PPR), where learned models and inference results are processed to obtain summary metrics, create dashboards, and power applications. We discuss specific operations in each of these components in Section 3. As we will demonstrate, these three components are generic and sufficient for describing a wide variety of supervised, semi-supervised, and unsupervised settings.

2.1 System Architecture

The HELIX system consists of a domain specific language (DSL) in Scala as the programming interface, a compiler for the DSL, and

an execution engine, as shown in Figure 1. The three components work collectively to *minimize the execution time for both the current iteration and subsequent iterations*:

1. Programming Interface (Section 3). HELIX provides a single Scala interface named `Workflow` for programming the entire workflow; the HELIX DSL also enables embedding of imperative code in declarative statements. Through just a handful of extensible operator types, the DSL supports a wide range of use cases for both data preprocessing and machine learning.

2. Compilation (Sections 4, 5.1–5.2). A `Workflow` is internally represented as a directed acyclic graph (DAG) of operator outputs. The DAG is compared to the one in previous iterations to determine reusability (Section 4). The *DAG Optimizer* uses this information to produce an optimal *physical execution plan* that *minimizes the one-shot runtime of the workflow*, by selectively loading previous results via a MAX-FLOW-based algorithm (Section 5.1–5.2).

3. Execution Engine (Section 5.3). The execution engine carries out the physical plan produced during the compilation phase, while communicating with the *materialization operator* to materialize intermediate results, to *minimize runtime of future executions*. The execution engine uses Spark [74] for data processing and domain-specific libraries for custom needs. HELIX defers operator pipelining and scheduling for asynchronous execution to Spark.

2.2 The Workflow Lifecycle

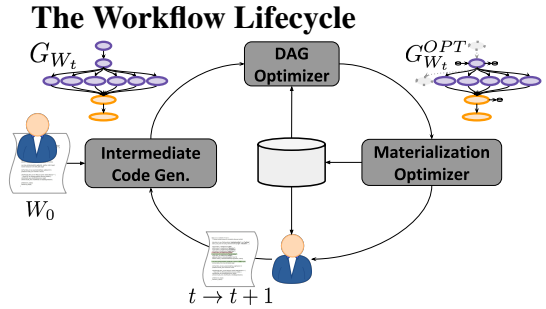

Figure 2: Roles of system components in the HELIX workflow lifecycle.

Given the system components described in the previous section, Figure 2 illustrates how they fit into the lifecycle of ML workflows. Starting with W_0, an initial version of the workflow, the lifecycle includes the following stages:

- **DAG Compilation.** The `Workflow` W_t is compiled into a DAG G_{W_t} of operator outputs.
- **DAG Optimization.** The DAG optimizer creates a physical plan $G^{OPT}_{W_t}$ to be executed by pruning and ordering the nodes in G_{W_t} and deciding whether any computation can be replaced with loading previous results from disk.
- **Materialization Optimization.** During execution, the materialization optimizer determines which nodes in $G^{OPT}_{W_t}$ should be persisted to disk for future use.
- **User Interaction.** Upon execution completion, the user may modify the workflow from W_t to W_{t+1} based on the results. The updated workflow W_{t+1} fed back to HELIX marks the beginning of a new iteration, and the cycle repeats.

2.3 Example Workflow

We demonstrate the usage of HELIX with a simple example ML workflow for predicting income using census data from Kohavi [29], shown in Figure 3a). We overlay the original workflow with an iterative update, with additions annotated with + and deletions annotated with −, while the rest of the lines are retained as is. We begin by describing the original workflow consisting of all the unannotated lines plus the line annotated with − (deletions). Additional details can be found in our technical report [70].

Original Workflow. A data collection `rows` is created from a data source `data` containing both training and test CSV files. The user declares simple features such as `age` and `education` corresponding to each column (lines 5–10), as well as discretized features like `ageBucket` (line 11), and interaction features for higher-order patterns (line 12). These features are then associated with `rows` (line 13), and used to create collection of examples `income` using `target` as labels (line 14). For L/I, the user declares an ML model named `incPred` (line 15), to be evaluated on all of the data in `income` (line 16). Subsequently, the user creates a new UDF named `checkResults` to check the prediction accuracy (line 17-18). Finally, the user declares that the output scalar named `checked` is to be computed from the test data in `income` (line 20) and is part of the output (line 21).

Original Workflow: Optimized DAG. The HELIX compiler translates the program in Figure 3a) into a DAG in Figure 3b), which is then transformed by the optimizer by eliminating `raceExt` since it does not contribute to the model. DPR steps are in purple, while others are in orange. Some of the intermediate results are materialized to disk (displayed with a drum).

Updated Workflow: Optimized DAG. In the updated version of the workflow, a new feature named `msExt` is added (below line 9), and `clExt` is removed (line 13); correspondingly, in the updated DAG, a new node is added for `msExt` (green edges), while `clExt` gets pruned (pink edges). In addition, HELIX chooses to load materialized results for `rows` from the previous iteration allowing `data` to be pruned, avoiding a costly parsing step. HELIX also loads `ageBucket` instead of recomputing the bucket boundaries requiring a full scan. HELIX materializes `predictions` in both iterations since it has changed. In this instance, HELIX is able to prune the computation of `ageBucket` and that of `rows` from `data`.

3. PROGRAMMING INTERFACE

To program ML workflows with high-level abstractions, HELIX users program in a language called HML, an *embedded DSL* in Scala. An embedded DSL exists as a library in the host language (Scala in our case), leading to seamless integration. LINQ [38], a data query framework integrated in .NET languages, is another example of an embedded DSL. In HELIX, users can freely incorporate Scala code for user-defined functions (UDFs) directly into HML. JVM-based libraries can be imported directly into HML to support application-specific needs. Development in other languages can be supported with wrappers in the same style as PySpark [55].

3.1 Operations in ML Workflows

Common operations in ML workflows can be decomposed into a small set of *basis functions* $\mathcal{F}$. Here, we simply introduce the members of $\mathcal{F}$. In our technical report [70], we show through a rigorous and extensive comparison that $\mathcal{F}$ covers all of the functionalities offered by Scikit-learn, thereby demonstrating coverage of common operations in ML workflows, as Scikit-learn is one of the most comprehensive ML libraries available.

Functions in $\mathcal{F}$ have natural analogs in both Scikit-learn ([70]) and HML (Section 3.2.2), thus serving as a mapping between the two programming interfaces. They can be grouped by the workflow components DPR, L/I, and PPR as follows:

DPR. The goal of DPR is to transform raw input data into learnable representations. DPR operations can be decomposed into the following categories:

- *Parsing*: transforming a record into a set of records, e.g., parsing an article into words via *tokenization*.

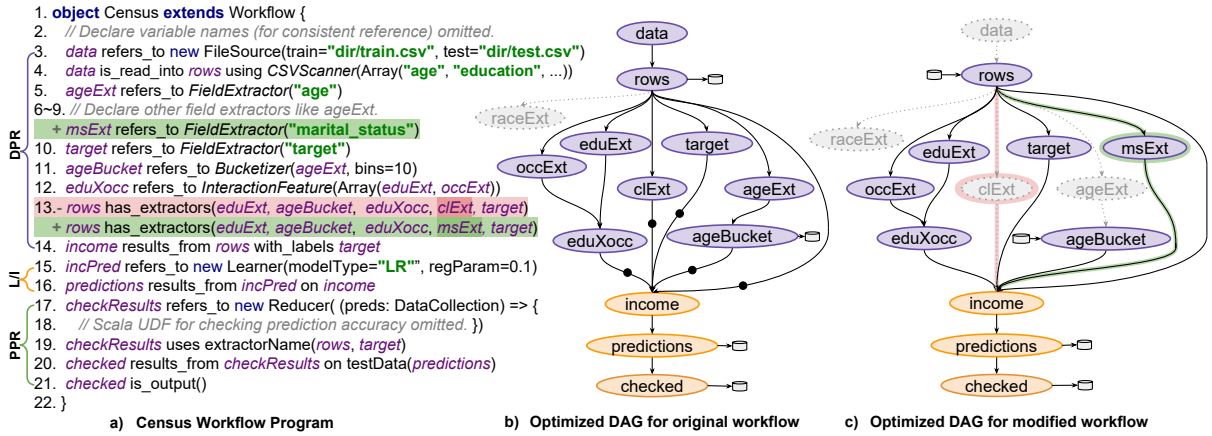

Figure 3: Example workflow for predicting income from census data.

- *Join*: combining multiple records into a single record, where r_i can come from different data sources.
- *Feature Extraction*: extracting features from a record.
- *Feature Transformation*: deriving a new set of features from the input features.
- *Feature Concatenation*: concatenating features extracted in separate operations to form an FV.

Note that sometimes these functions need to be *learned* from the input data. For example, discretizing a continuous feature x_i into four even-sized bins requires the distribution of x_i. We address this use case along with L/I next.

L/I. L/I encompasses learning both ML models and feature transformation functions mentioned above. Note that while applying feature transformations is part of DPR, learning the functions themselves is in L/I. Complex ML tasks can be broken down into simple learning steps captured by these two operations, e.g., image captioning can be broken down into object identification via classification, followed by sentence generation using a language model [27]. Thus, L/I can be decomposed into:

- *Learning*: learning a function f from the input dataset.
- *Inference*: using the ML model f to infer feature values.

PPR. A wide variety of operations can take place in PPR, including model evaluation and data visualization. At a high-level, they all share the same function:

- *Reduce*: applying an operation on the input dataset(s) and non-dataset object(s). For example, for an input dataset and a specific feature name as the non-dataset object, we can produce visualizations of the feature values collected over the input dataset.

3.2 HML

HML is a declarative language for specifying an ML workflow DAG. The basic building blocks of HML are HELIX *objects*, which correspond to the nodes in the DAG. Each HELIX object is either a *data collection* (DC) or an *operator*. Statements in HML either declare new instances of objects or relationships between declared objects. Users program the entire workflow in a single `Workflow` interface, as shown in Figure 3a). The complete grammar for HML in Backus-Naur Form as well as the semantics of all of the expressions can be found in the technical report [70]. Here, we describe high-level concepts including DCs and operators and discuss the strengths and limitations of HML in Section 3.3.

3.2.1 Data Collections

A *data collection* (DC) is analogous to a relation in a RDBMS; each *element* in a DC is analogous to a tuple. The content of a DC either derives from disk, e.g., `data` in Line 3 in Figure 3a), or from operations on other DCs, e.g., `rows` in Line 4 in Figure 3a). An element in a DC can either be a *semantic unit*, the data structure for DPR, or an *example*, the data structure for L/I.

A DC can only contain a single type of element. DC_{SU} and DC_{E} denote a DC of semantic units and a DC of examples, respectively. The type of elements in a DC is determined by the operator that produced the DC and not explicitly specified by the user. We elaborate on the relationship between operators and element types in Section 3.2.2, after introducing the operators.

Semantic units. Many DPR operations require going through the entire dataset to learn the exact transformation or extraction function. For a workflow with many such operations, processing $\mathcal{D}$ to learn each operator separately can be highly inefficient. We introduce the notion of semantic units (SU) to compartmentalize the logical and physical representations of features, so that the learning of DPR functions can be delayed and batched.

Formally, each SU contains an input i, which can be a set of records or FVs, a pointer to a DPR function f, which can be of type parsing, join, feature extraction, feature transformation, or feature concatenation, and an output o, which can be a set of records or FVs and is the output of f on i. The variables i and f together serve as the *semantic*, or logical, representation of the features, whereas o is the lazily evaluated physical representation that can only be obtained after f is fully instantiated.

Examples. Examples gather all the FVs contained in the output of various SUs into a single FV for learning. Formally, an example contains a set of SUs S, and an optional pointer to one of the SUs whose output will be used as the label in supervised settings, and an output FV, which is formed by concatenating the outputs of S.

Sparse vs. Dense Features. The combination of SUs and examples affords HELIX a great deal of flexibility in the physical representation of features. Users can explicitly program their DPR functions to output dense vectors, in applications such as computer vision. For sparse categorical features, they are kept in the raw key-value format until the final FV assembly, where they are transformed into sparse or dense vectors depending on whether the ML algorithm supports sparse representations. When assembling a mixture of dense and spare FVs, HELIX currently opts for a dense representation but can be extended to support optimizations considering space and time tradeoffs.

Unified learning support. HML provides unified support for training and test data by treating them as a single DC, as done in Line 4 in Figure 3a). This design ensures that both training and test

data undergo the exact same data preprocessing steps, eliminating bugs caused by inconsistent data preprocessing procedures handling training and test data separately. HELIX automatically selects the appropriate data for training and evaluation. However, if desired, users can handle training and test data differently by specifying separate DAGs for training and testing. Common operators can be shared across the two DAGs without code duplication.

3.2.2 Operators

Operators in HELIX are designed to cover the functions enumerated in Section 3.1, using the data structures introduced above. A HELIX *operator* takes one or more DCs and outputs DCs, ML models, or scalars. Each operator encapsulates a function f, written in Scala, to be applied to individual elements in the input DCs. We describe the relationships between operator interfaces in HML and $\mathcal{F}$ enumerated in Section 3.1 below.

Scanner. *Scanner* is the interface for parsing $\in \mathcal{F}$ and acts like a flatMap, i.e., for each input element, it adds zero or more elements to the output DC. The input and output of Scanner are DC_{SU}s. `CSVScanner` in Line 4 of Figure 3a) is an example of a Scanner that parses lines in a CSV file into key-value pairs for columns.

Synthesizer. *Synthesizer* supports join $\in \mathcal{F}$, for elements both across multiple DCs and within the same DC. Thus, it can also support aggregation operations such as sliding windows in time series. Synthesizers also serve the important purpose of specifying the set of SUs that make up an example . In the simple case where each SU in a DC_{SU} corresponds to an example, a pass-through synthesizer is implicitly declared by naming the output DC_E, such as in Line 14 of Figure 3a).

Learner. *Learner* is the interface for learning and inference $\in \mathcal{F}$, in a single operator. A learner operator L contains a learned function f, which can be an ML model or a feature transformation function learned from the input dataset. When f is empty, L learns a model using input data designated for model training; when f is populated, L performs inference on the input data using f and outputs the inference results into a DC_E. For example, the learner `incPred` in Line 15 of Figure 3a) is a learner trained on the "train" portion of the DC_E `income` and outputs inference results as the DC_E `predictions`.

Extractor. *Extractor* is the interface for feature extraction and feature transformation $\in \mathcal{F}$. Extractor contains the function f applied on the input of SUs, thus the input and output to an extractor are DC_{SU}s. For functions that need to be learned from data, Extractor contains a pointer to the learner operator for learning f.

Reducer. Reducer is the interface for reduce $\in \mathcal{F}$ and thus the main operator interface for PPR. The inputs to a reducer are DC_E and an optional scalar and the output is a scalar, where scalars refer to non-dataset objects. For example, `checkResults` in Figure 3a) Line 17 is a reducer that computes the prediction accuracy of the inference results in `predictions`.

3.3 Scope and Limitations

Coverage. We showed in Section 3.2 that HML captures all functions in $\mathcal{F}$, which is shown to cover all major operations in ML workflows [70]. While HML's interfaces are general enough to support all the common use cases, users can additionally manually plug into our interfaces external implementations, such as from MLLib [39] and Weka [25], of missing operations. *Note that we provide utility functions that allow functions to work directly with raw records and FVs instead of HML data structures to enable direct application of external libraries.* We demonstrate in Section 6 that the current set of implemented operations is sufficient for supporting applications across different domains.

Limitations. Since HELIX currently relies on its Scala DSL for workflow specification, popular non-JVM libraries, such as TensorFlow [4] and Pytorch [46], cannot be imported easily without significantly degrading performance compared to their native runtime environment. That said, our contributions in materialization and reuse apply across all languages. In the future, we plan on abstracting the DAG representation in HELIX into a language-agnostic system that can sit below the language layer for all DAG based systems, including TensorFlow, Scikit-learn, and Spark.

The other downside of HML is that ML models are treated largely as black boxes. Thus, work on optimizing learning, e.g., [52, 77], orthogonal to (and can therefore be combined with) our work, which operates at a coarser granularity.

4. COMPILATION & REPRESENTATION

In this section, we describe the `Workflow` DAG, the abstract model used internally by HELIX to represent a `Workflow` program. The `Workflow` DAG model enables operator-level change tracking between iterations and end-to-end optimizations.

4.1 The Workflow DAG

At compile time, HELIX's intermediate code generator constructs a *`Workflow` DAG* from HML declarations.

Definition 1. *For a `Workflow` W containing* HELIX *operators $F = \{f_i\}$, the `Workflow` DAG is a directed acyclic graph $G_W = (N, E)$, where node $n_i \in N$ represents the output of $f_i \in F$ and $(n_i, n_j) \in E$ if the output of f_i is an input to f_j.*

Figure 3b) shows the `Workflow` DAG for the program in Figure 3a). This transformation is straightforward, creating a node for each declared operator and adding edges between nodes based on the linking expressions, e.g., `A results_from B` creates an edge (B, A). Additionally, the intermediate code generator introduces edges not specified in the `Workflow` between the extractor and the synthesizer nodes, such as the edges marked by dots (•) in Figure 3b). These edges connect extractors to downstream DCs in order to automatically aggregate all features for learning. One concern is that this may lead to redundant computation of unused features; we describe pruning mechanisms to address this issue in Section 5.4.

4.2 Tracking Changes

As described in Section 2.2, a user starts with an initial workflow W_0 and iterates on this workflow. Let W_t be the version of the workflow at iteration $t \geq 0$ with the corresponding DAG $G_W^t = (N_t, E_t)$; W_{t+1} denotes the workflow obtained in the next iteration. To describe the changes between W_t and W_{t+1}, we introduce the notion of *equivalence*.

Definition 2. *A node $n_i^t \in N_t$ is* equivalent *to $n_i^{t+1} \in N_{t+1}$, denoted as $n_i^t \equiv n_i^{t+1}$, if **a**) the operators corresponding to n_i^t and n_i^{t+1} compute identical results on the same inputs and **b**) $n_j^t \equiv n_j^{t+1}$ $\forall$ $n_j^t \in parents(n_i^t), n_j^{t+1} \in parents(n_i^{t+1})$. We say $n_i^{t+1} \in N_{t+1}$ is* original *if it has no equivalent node in N_t.*

Equivalence is symmetric, i.e., $n_i^{t'} \equiv n_i^t \Leftrightarrow n_i^t \equiv n_i^{t'}$, and transitive, i.e., $(n_i^t \equiv n_i^{t'} \wedge n_i^{t'} \equiv n_i^{t''}) \Rightarrow n_i^t \equiv n_i^{t''}$. Newly added operators in W_{t+1} do not have equivalent nodes in W_t; neither do nodes in W_t that are removed in W_{t+1}. For a node that persists across iterations, we need both the operator and the ancestor nodes to stay the same for equivalence. Using this definition of equivalence, we determine if intermediate results on disk can be safely reused through the notion of equivalent materialization:

Definition 3. *A node $n_i^t \in N_t$ has an* equivalent materialization *if $n_i^{t'}$ is stored on disk, where $t' \leq t$ and $n_i^{t'} \equiv n_i^t$.*

One challenge in determining equivalence is deciding whether two versions of an operator compute the same results on the same input. For arbitrary functions, this is undecidable as proven by Rice's Theorem [54]. The programming language community has a large body of work on verifying operational equivalence for specific classes of programs [66, 49, 23]. HELIX currently employs a simple representational equivalence verification—an operator remains equivalent across iterations if its declaration in the DSL is not modified and all of its ancestors are unchanged. Incorporating more advanced techniques for verifying equivalence is future work.

To guarantee correctness (the proof for which is in our technical report [70]), i.e., results obtained at iteration t reflect the specification for W_t and are computed from the appropriate input, we impose the constraint:

Constraint 1. *At iteration $t+1$, if an operator n_i^{t+1} is original, it must be recomputed.*

5. OPTIMIZATION

In this section, we describe HELIX's workflow-level optimizations, motivated by the observation that *workflows often share a large amount of intermediate computation between iterations*; thus, if certain intermediate results are materialized at iteration t, these can be used at iteration $t+1$. We identify two distinct sub-problems: OPT-EXEC-PLAN, which selects the operators to reuse given previous materializations (Section 5.2), and OPT-MAT-PLAN, which decides what to materialize to accelerate future iterations (Section 5.3). We finally discuss pruning optimizations to eliminate redundant computations (Section 5.4). We begin by introducing common notation and definitions.

5.1 Preliminaries

When introducing variables below, we drop the iteration number t from W_t and G_W^t when we are considering a static workflow.

Operator Metrics. In a Workflow DAG $G_W = (N, E)$, each node $n_i \in N$ corresponding to the output of the operator f_i is associated with a compute time c_i, the time it takes to compute n_i from inputs in memory. Once computed, n_i can be materialized on disk and loaded back in subsequent iterations in time l_i, referred to as its *load time*. If n_i does not have an equivalent materialization as defined in Definition 3, we set $l_i = \infty$. Root nodes in the Workflow DAG, which correspond to data sources, have $l_i = c_i$.

Operator State. During the execution of workflow W, each node n_i assumes one of the following states:

- *Load*, or S_l, if n_i is loaded from disk;
- *Compute*, or S_c, n_i is computed from inputs;
- *Prune*, or S_p, if n_i is skipped (neither loaded nor computed).

Let $s(n_i) \in \{S_l, S_c, S_p\}$ denote the state of each $n_i \in N$. To ensure that nodes in the Compute state have their inputs available, i.e., not *pruned*, the states in a Workflow DAG $G_W = (N, E)$ must satisfy the following *execution state constraint*:

Constraint 2. *For a node $n_i \in N$, if $s(n_i) = S_c$, then $s(n_j) \neq S_p$ for every $n_j \in parents(n_i)$.*

Workflow Run Time. A node n_i in state S_c, S_l, or S_p has run time c_i, l_i, or 0, respectively. The total run time of W w.r.t. s is thus

$$T(W, s) = \sum_{n_i \in N} \mathbb{I}\{s(n_i) = S_c\}\, c_i + \mathbb{I}\{s(n_i) = S_l\}\, l_i \qquad (1)$$

where $\mathbb{I}\{\}$ is the indicator function.

Clearly, setting all nodes to S_p trivially minimizes Equation 1. However, recall that Constraint 1 requires all original operators to be rerun. Thus, if an original operator n_i is introduced, we must have $s(n_i) = S_c$, which by Constraint 2 requires that $S(n_j) \neq S_p \; \forall n_j \in parents(n_i)$. We explore how to determine the states for each nodes to minimize Equation 1 next.

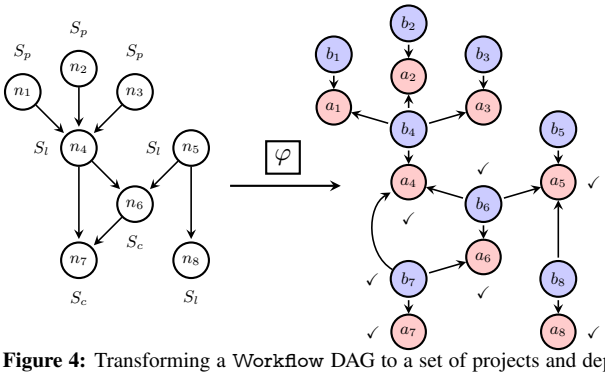

Figure 4: Transforming a Workflow DAG to a set of projects and dependencies. Checkmarks (✓) in the RHS DAG indicate a feasible solution to PSP, which maps onto the node states (S_p, S_c, S_l) in the LHS DAG.

5.2 Optimal Execution Plan

The *Optimal Execution Plan* (OEP) problem is the core problem solved by HELIX's DAG optimizer, which determines at compile time the optimal execution plan given results and statistics from previous iterations.

Problem 1. *(*OPT-EXEC-PLAN*) Given a Workflow W with DAG $G_W = (N, E)$, the compute time and the load time c_i, l_i for each $n_i \in N$, and a set of previously materialized operators M, find a state assignment $s : N \rightarrow \{S_c, S_l, S_p\}$ that minimizes $T(W, s)$ while satisfying Constraint 1 and Constraint 2.*

Let $T^*(W)$ be the minimum execution time achieved by the solution to OEP, i.e.,

$$T^*(W) = \min_s T(W, s) \qquad (2)$$

Since this optimization takes place *prior* to execution, we must resort to operator statistics from past iterations. *This does not compromise accuracy because if a node n_i has an equivalent materialization as defined in Definition 2, we would have run the exact same operator before and recorded accurate c_i and l_i.* A node n_i without an equivalent materialization, such as a model with changed hyperparameters, needs to be recomputed (Constraint 1).

Deciding to load certain nodes can have cascading effects since ancestors of a loaded node can potentially be pruned, leading to large reductions in run time. On the other hand, Constraint 2 disallows the parents of computed nodes to be pruned. Thus, the decisions to load a node n_i can be affected by nodes outside of the set of ancestors to n_i. For example, in the DAG on the left in Figure 4, loading n_7 allows n_{1-6} to be potentially pruned. However, the decision to compute n_8, possibly arising from the fact that $l_8 \gg c_8$, requires that n_5 must not be pruned.

Despite such complex dependencies between the decisions for individual nodes, Problem 1 can be solved optimally in polynomial time through a linear time reduction to the *project-selection problem* (PSP), which is an application of MAX-FLOW [28].

Problem 2. PROJ-SELECTION-PROBLEM *(PSP) Let P be a set of projects. Each project $i \in P$ has a real-valued profit p_i and a set of prerequisites $Q \subseteq P$. Select a subset $A \subseteq P$ such that all prerequisites of a project $i \in A$ are also in A and the total profit of the selected projects, $\sum_{i \in A} p_i$, is maximized.*

Reduction to the Project Selection Problem. We can reduce an instance of Problem 1 x to an equivalent instance of PSP y such that the optimal solution to y maps to an optimal solution of x. Let $G =$

Algorithm 1: OEP via Reduction to PSP

```
   Input: G_W = (N, E), {l_i}, {c_i}
1  P ← ∅;
2  for n_i ∈ N do
3      P ← P ∪ {a_i} ;                          // Create a project a_i
4      profit[a_i] ← −l_i ;                     // Set profit of a_i to −l_i
5      P ← P ∪ {b_i} ;                          // Create a project b_i
6      profit[b_i] ← l_i − c_i ;                // Set profit of b_i to l_i − c_i
       // Add a_i as prerequisite for b_i.;
7      prerequisite[b_i] ← prerequisite[b_i] ∪ a_i;
8      for (n_i, n_j) ∈ {edges leaving from n_i} ⊆ E do
           // Add a_i as prerequisite for b_j.;
9          prerequisite[b_j] ← prerequisite[b_j] ∪ a_i;
   // A is the set of projects selected by PSP;
10 A ← PSP(P, profit[], prerequisite[]);
11 for n_i ∈ N do                               // Map PSP solution to node states
12     if a_i ∈ A and b_i ∈ A then
13         s[n_i] ← S_c;
14     else if a_i ∈ A and b_i ∉ A then
15         s[n_i] ← S_l;
16     else
17         s[n_i] ← S_p;
18 return s[] ;                                 // State assignments for nodes in G_W.
```

(N, E) be the Workflow DAG in x, and P be the set of projects in y. We can visualize the prerequisite requirements in y as a DAG with the projects as the nodes and an edge (j, i) indicating that project i is a prerequisite of project j. The reduction, φ, depicted in Figure 4 for an example instance of x, is shown in Algorithm 1. For each node $n_i \in N$, we create two projects in PSP: a_i with profit $-l_i$ and b_i with profit $l_i - c_i$. We set a_i as the prerequisite for b_i. For an edge $(n_i, n_j) \in E$, we set the project a_i corresponding to node n_i as the prerequisite for the project b_j corresponding to node n_j. Selecting both projects a_i and b_i corresponding to n_i is equivalent to computing n_i, i.e., $s(n_i) = S_c$, while selecting only a_i is equivalent to loading n_i, i.e., $s(n_i) = S_l$. Nodes with neither projects selected are pruned. An example solution mapping from PSP to OEP is shown in Figure 4. Projects $a_4, a_5, a_6, b_6, a_7, b_7, a_8$ are selected, which cause nodes n_4, n_5, n_8 to be loaded, n_6 and n_7 to be computed, and n_1, n_2, n_3 to be pruned.

Overall, the optimization objective in PSP models the "savings" in OEP incurred by loading nodes instead of computing them from inputs. We create an equivalence between cost minimization in OEP and profit maximization in PSP by mapping the costs in OEP to negative profits in PSP. For a node n_i, picking only project a_i (equivalent to loading n_i) has a profit of $-l_i$, whereas picking both a_i and b_i (equivalent to computing n_i) has a profit of $-l_i + (l_i - c_i) = -c_i$. The prerequisites established in Line 7 that require a_i to also be picked if b_i is picked are to ensure correct cost to profit mapping. The prerequisites established in Line 9 corresponds to Constraint 2. For a project b_i to be picked, we must pick every a_j corresponding to each parent n_j of n_i. If it is impossible ($l_j = \infty$) or costly to load n_j, we can offset the load cost by picking b_j for computing n_j. However, computing n_j also requires its parents to be loaded or computed, as modeled by the outgoing edges from b_j. The fact that a_i projects have no outgoing edges corresponds to the fact loading a node removes its dependency on all ancestor nodes.

Theorem 1. *Given an instance of* OPT-EXEC-PLAN *x, the reduction in Algorithm 1 produces a feasible and optimal solution to x.*

See the technical report [70] for a proof.

5.3 Optimal Materialization Plan

The OPT-MAT-PLAN (OMP) problem is tackled by HELIX's materialization optimizer: while running workflow W_t at iteration t, intermediate results are selectively materialized for the purpose of accelerating execution in iterations $> t$. We now formally introduce OMP and show that it is NP-HARD even under strong assumptions. We propose an online heuristic for OMP that runs in linear time and achieves good reuse rate in practice (as we will show in Section 6), in addition to minimizing memory footprint by avoiding unnecessary caching of intermediate results.

Materialization cost. We let s_i denote the *storage cost* for materializing n_i, representing the size of n_i on disk. When loading n_i back from disk to memory, we have the following relationship between load time and storage cost: $l_i = s_i/(\text{disk read speed})$. For simplicity, we also assume the time to write n_i to disk is the same as the time for loading it from disk, i.e., l_i. We can easily generalize to the setting where load and write latencies are different.

To quantify the benefit of materializing intermediate results at iteration t on subsequent iterations, we formulate the *materialization run time* $T_M(W_t)$ to capture the tradeoff between the additional time to materialize intermediate results and the run time reduction in iteration $t + 1$. Although materialized results can be reused in multiple future iterations, we only consider the $(t + 1)$th iteration since the total number of future iterationsis unknown.

Definition 4. *Given a workflow W_t, operator metrics c_i, l_i, s_i for every $n_i \in N_t$, and a subset of nodes $M \subseteq N_t$, the* materialization run time *is defined as*

$$T_M(W_t) = \sum_{n_i \in M} l_i + T^*(W_{t+1}) \tag{3}$$

where $\sum_{n_i \in M} l_i$ is the time to materialize all nodes selected for materialization, and $T^*(W_t)$ is the optimal workflow run time obtained using the algorithm in Section 5.2, with M materialized.

Equation 3 defines the optimization objective for OMP.

Problem 3. *(*OPT-MAT-PLAN*) Given a* Workflow *W_t with DAG $G_W^t = (N_t, E_t)$ at iteration t and a storage budget S, find a subset of nodes $M \subseteq N_t$ to materialize at t in order to minimize $T_M(W_t)$, while satisfying the storage constraint $\sum_{n_i \in M} s_i \leq S$.*

As discussed in [69], there are many possibilities for W_{t+1}, and they vary by application domain. User modeling and predictive analysis of W_{t+1} itself is a substantial research topic that we will address in future work. This user model can be incorporated into OMP by using the predicted changes to better estimate the likelihood of reuse for each operator. However, even under very restrictive assumptions about W_{t+1}, we can show that OPT-MAT-PLAN is NP-HARD, via a simple reduction from the KNAPSACK problem.

Theorem 2. OPT-MAT-PLAN *is NP-hard.*

See the technical report [70] for a proof.

Streaming constraint. Even when W_{t+1} is known, solving OPT-MAT-PLAN optimally requires knowing the run time statistics for all operators, which can be fully obtained only at the end of the workflow. Deferring materialization decisions until the end requires all intermediate results to be cached or recomputed, which imposes undue pressure on memory and cripples performance. Unfortunately, reusing statistics from past iterations as in Section 5.2 is not viable here because of the cold-start problem—materialization decisions need to be made for new operators based on realistic statistics. Thus, to avoid slowing down execution with high memory usage, we impose the following constraint.

Definition 5. *Given a* Workflow *DAG $G_w = (N, E)$, $n_i \in N$ is* out-of-scope *at runtime if all children of n_i have been computed or reloaded from disk, thus removing all dependencies on n_i.*

Constraint 3. *Once n_i becomes out-of-scope, it is either materialized immediately or removed from cache.*

OMP Heuristics. We now describe the heuristic employed by HELIX to approximate OMP while satisfying Constraint 3.

Algorithm 2: Streaming OMP

Data: $G_w = (N, E)$, $\{l_i\}$, $\{c_i\}$, $\{s_i\}$, storage budget S
$M \leftarrow \varnothing$;
while *Workflow is running* **do**
  $O \leftarrow$ FindOutOfScope(N);
  for $n_i \in O$ **do**
    if $C(n_i) > 2l_i$ *and* $S - s_i \geq 0$ **then**
      Materialize n_i;
      $M \leftarrow M \cup \{n_i\}$;
      $S \leftarrow S - s_i$

Definition 6. *Given* `Workflow` *DAG* $G_w = (N, E)$*, the* cumulative run time *for a node* n_i *is defined as*

$$C(n_i) = t(n_i) + \sum_{n_j \in ancestors(n_i)} t(n_j) \tag{4}$$

where $t(n_i) = \mathbb{I}\{s(n_i) = S_c\}\, c_i + \mathbb{I}\{s(n_i) = S_l\}\, l_i$.

Algorithm 2 shows the heuristics employed by HELIX's materialization optimizer to decide what intermediate results to materialize. In essence, Algorithm 2 decides to materialize if twice the load cost is less than the cumulative run time for a node. The intuition behind this algorithm is that assuming loading a node allows all of its ancestors to be pruned, the materialization time in iteration t and the load time in iteration $t + 1$ combined should be less than the total pruned compute time, for the materialization to be cost effective.

Due to the streaming Constraint 3, complex dependencies between descendants of ancestors such as the one between n_5 and n_8 in Figure 4 previously described in Section 5.2, are ignored by Algorithm 2—we cannot retroactively update our decision for n_5 after n_8 has been run. We show in Section 6 that this simple algorithm is effective in multiple application domains.

5.4 Workflow DAG Pruning

In addition to optimizations involving intermediate result reuse, HELIX further reduces overall workflow execution time by pruning extraneous operators from the `Workflow` DAG. In a nutshell, HELIX traverses the DAG backwards from the output nodes and prunes away any nodes not visited in this traversal—a simple form of *program slicing* [65]. HELIX provides two additional mechanisms for pruning operators other than using the lack of output dependency: 1) user can explicitly specify nodes to be excluded in HML for manual feature selection; and 2) if the user desires it, HELIX can inspect relevant data to determine low-impact operators that can be removed without compromising the model performance. We plan on investigating the latter extensively in future work.

6. EMPIRICAL EVALUATION

The goal of our evaluation is to test if HELIX 1) *supports* ML workflows in a variety of application domains; 2) *accelerates* iterative execution through intermediate result reuse, compared to other ML systems that don't optimize iteration; 3) is *efficient*, enabling optimal reuse without incurring a large storage overhead.

6.1 Systems and Baselines for Comparison

We compare the optimized version of HELIX, HELIX OPT, against two state-of-the-art ML workflow systems: KeystoneML [56], and DeepDive [75]. In addition, we compare HELIX OPT with two simpler versions, HELIX AM and HELIX NM. While we compare against DeepDive, and KeystoneML to verify 1) and 2) above, HELIX AM and HELIX NM are used to verify 3). We describe each of these variants below:

KeystoneML. KeystoneML [56] is a system, written in Scala and built on top of Spark, for the construction of large scale, end-to-end, ML pipelines. KeystoneML specializes in classification tasks on structured input data. No intermediate results are materialized in KeystoneML, as it does not optimize execution across iterations.

DeepDive. DeepDive [75, 17] is a system, written using Bash scripts and Scala for the main engine, with a database backend, for the construction of end-to-end information extraction pipelines. Additionally, DeepDive provides limited support for classification tasks. All intermediate results are materialized in DeepDive.

HELIX **OPT**. A version of HELIX that uses Algorithm 1 for the optimal reuse strategy and Algorithm 2 to decide what to materialize.

HELIX **AM**. A version of HELIX that uses the same reuse strategy as HELIX OPT and *always materializes* all intermediate results.

HELIX **NM**. A version of HELIX that uses the same reuse strategy as HELIX OPT and *never materializes* any intermediate results.

6.2 Workflows

We conduct our experiments using four real-world ML workflows spanning a range of application domains. Table 1 summarizes the characteristics of the four workflows, described next. We are interested in four properties when characterizing each workflow:

- *Number of data sources:* whether the input data comes from a single source (e.g., a CSV file) or multiple sources (e.g., documents and a knowledge base), necessitating joins.
- *Input to example mapping:* the mapping from each input data unit (e.g., a line in a file) to each learning example for ML. One-to-many mappings require more complex data preprocessing than one-to-one mappings.
- *Feature granularity:* fine-grained features involve applying extraction logic on a specific piece of the data (e.g., 2nd column) and are often application-specific, whereas coarse-grained features are obtained by applying an operation, usually a standard DPR technique such as normalization, on the entire dataset.
- *Learning task type:* while classification and structured prediction tasks both fall under supervised learning for having observed labels, structured prediction workflows involve more complex data preprocessing and models; unsupervised learning tasks do not have known labels, so they often require more qualitative and fine-grained analyses of outputs.

Census Workflow. This workflow corresponds to a classification task with simple features from structured inputs from the DeepDive Github repository [1]. It uses the Census Income dataset [19], with 14 attributes representing demographic information, with the goal to predict whether a person's annual income is >50K, using fine-grained features derived from input attributes. The complexity of this workflow is representative of use cases in the social and natural sciences, where covariate analysis is conducted on well-defined variables. HELIX code for the initial version of this workflow is shown in Figure 3a).

Genomics Workflow. This workflow is described in Example 1, involving two major steps: 1) split the input articles into words and learn vector representations for entities of interest, identified by joining with a genomic knowledge base, using word2vec [40]; 2) cluster the vector representation of genes using K-Means to identify functional similarity. This workflow has minimal data preprocessing with no specific features but involves multiple learning steps. Both learning steps are unsupervised, which leads to more qualitative and exploratory evaluations of the model outputs. We include a workflow with unsupervised learning and multiple learning steps to verify that the system is able to accommodate variability in the learning task.

Information Extraction (IE) Workflow. This workflow involves identifying mentions of spouse pairs from news articles, using a knowledge-base of known spouse pairs, from DeepDive [17]. The

Table 1: Summary of workflow characteristics and support by the systems compared. Grayed out cells indicate that the system in the row does not support the workflow in the column. ✓∗ indicates that the implementation is by the original developers of DeepDive/KeystoneML.

	Census (Source: [1])	**Genomics** (Source: [53])	**IE** (Source: [17])	**MNIST** (Source: [56])
Num. Data Source	Single	Multiple	Multiple	Single
Input to Example Mapping	One-to-One	One-to-Many	One-to-Many	One-to-One
Feature Granularity	Fine Grained	N/A	Fine Grained	Coarse Grained
Learning Task Type	Supervised; Classification	Unsupervised	Structured Prediction	Supervised; Classification
Application Domain	Social Sciences	Natural Sciences	NLP	Computer Vision
Supported by HELIX	✓	✓	✓	✓
Supported by KeystoneML	✓	✓		✓∗
Supported by DeepDive	✓∗		✓∗	

objective is to extract structured information from unstructured input text, using complex fine-grained features such as part-of-speech tagging. This workflow exemplifies use cases in information extraction, and tests a system's ability to handle joins and complex data preprocessing.

MNIST Workflow. The MNIST dataset [34] contains images of handwritten digits to be classified, which is a well-studied task in the computer vision community, from the KeystoneML [56] evaluation. The workflow involves nondeterministic (and hence not reusable) data preprocessing, with a substantial fraction of the overall run time spent on L/I in a typical iteration. We include this application to ensure that in the extreme case where there is little reuse across iterations, HELIX does not incur a large overhead.

Each workflow was implemented in HELIX, and if supported, in DeepDive and KeystoneML, with ✓* in Table 1 indicating that we used an existing implementation by the developers of DeepDive or KeystoneML, with scripts enumerated in [70]. DeepDive has its own DSL, while KeystoneML's programming interface is an embedded DSL in Scala, similar to HML. We explain limitations that prevent DeepDive and KeystoneML from supporting certain workflows (grey cells) in Section 6.5.1.

6.3 Running Experiments

Simulating iterative development. In our experiments, we modify the workflows to simulate typical iterative development by a ML application developer or data scientist. Instead of arbitrarily choosing operators to modify in each iteration, we use the iteration frequency in Figure 3 from our literature study [69] to determine the type of modifications to make in each iteration, for the specific domain of each workflow. We convert the iteration counts into fractions that represent the likelihood of a certain type of change. At each iteration, we draw an iteration type from {DPR, L/I, PPR} according to these likelihoods. Then, we randomly choose an operator of the drawn type and modify its source code. For example, if an "L/I" iteration were drawn, we might change the regularization parameter for the ML model. We run 10 iterations per workflow (except NLP, which has only DPR iterations).

Environment. All single-node experiments are run on a server with 125 GiB of RAM, 16 cores on 8 CPUs (Intel Xeon @ 2.40GHz), and 2TB HDD with 170MB/s as both the read and write speeds. Distributed experiments are run on nodes each with 64GB of RAM, 16 cores on 8 CPUs (Intel Xeon @ 2.40GHz), and 500GB of HDD with 180MB/s as both the read and write speeds. We set the storage budget in HELIX to 10GB. That is, 10GB is the maximum accumulated disk storage for HELIX OPT at all times during the experiments. After running the initial version to obtain the run time for iteration 0, a workflow is modified according to the type of change determined as above. In all four systems the modified workflow is recompiled. In DeepDive, we rerun the workflow using the command `deepdive run`. In HELIX and KeystoneML, we resubmit a job to Spark in local mode. Although HELIX and KeystoneML support distributed execution via Spark, DeepDive needs to run on a single server. Thus, we compare against all systems on a single node and additionally compare against KeystoneML on clusters.

6.4 Metrics

We evaluate each system's ability to support diverse ML tasks by qualitative characterization of the workflows and use-cases supported by each system. Our primary metric for workflow execution is *cumulative run time* over multiple iterations. When computing cumulative run times, we average the per-iteration run times over five complete runs for stability. Note that the per-iteration time measures both the time to execute the workflow and any time spent to materialize intermediate results. We also measure *memory usage* (results shown in the technical report) and measure *storage size* to compare the run time reduction to storage ratio of time-efficient approaches. Storage is compared only for variants of HELIX since other systems do not support automatic reuse.

6.5 Evaluation vs. State-of-the-art Systems

6.5.1 Use Case Support

HELIX supports ML workflows in multiple distinct application domains, spanning tasks with varying complexity in both supervised and unsupervised learning.

Recall that the four workflows used in our experiments are in social sciences, NLP, computer vision, and natural sciences, respectively. Table 1 lists the characteristics of each workflow and the three systems' ability to support it. Both KeystoneML and DeepDive have limitations that prevent them from supporting certain types of tasks. The pipeline programming model in KeystoneML is effective for large scale classification and can be adapted to support unsupervised learning. However, it makes fine-grained features cumbersome to program and is not conducive to structured prediction tasks due to complex data preprocessing. On the other hand, DeepDive is highly specialized for information extraction and focuses on supporting data preprocessing. Unfortunately, its learning and evaluation components are not configurable by the user, limiting the type of ML tasks supported. DeepDive is therefore unable to support the MNIST and genomics workflows, both of which required custom ML models. Additionally, we are only able to show DeepDive performance for DPR iterations for the supported workflows in our experiments.

6.5.2 Cumulative Run Time

HELIX achieves up to **19**× cumulative run time reduction in ten iterations over state-of-the-art ML systems.

Figure 5 shows the cumulative run time for all four workflows. The x-axis shows the iteration number, while the y-axis shows the cumulative run time in log scale at the ith iteration. Each point represents the cumulative run time of the first i iterations. The color under the curve indicates the workflow component modified in each iteration (purple = DPR, orange = L/I, green = PPR). For example, the DPR component was modified in the first iteration of Census.

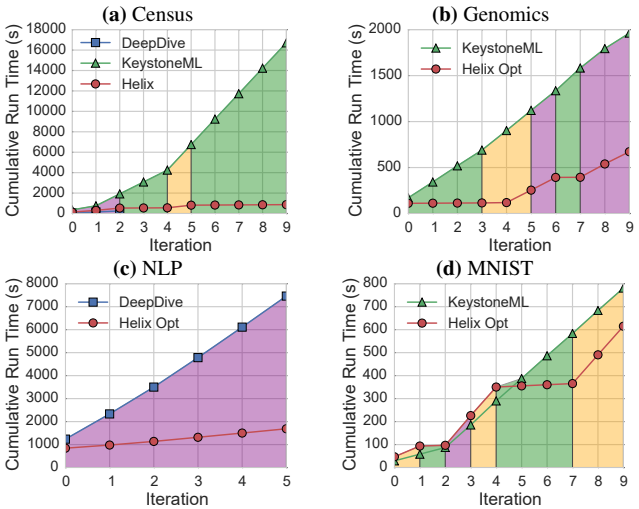

Figure 5: Cumulative run time for the four workflows. The color under the curve indicates the type of change in each iteration: purple for DPR, orange for L/I, and green for PPR.

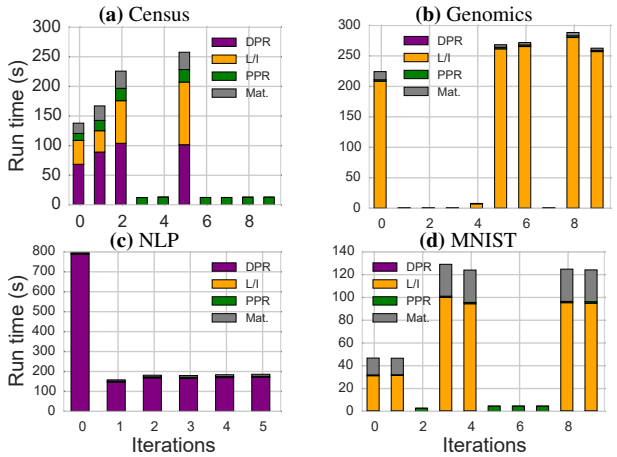

Figure 6: Run time breakdown by workflow component and materialization time per iteration for HELIX.

Figure 6 shows the breakdown by workflow components and materialization for the individual iteration run times in HELIX, with the same color scheme as in Figure 5 for the workflow components and gray for materialization time.

Census. As shown in Figure 5(a), the census workflow has the largest cumulative run time gap between HELIX OPT and the competitor systems—HELIX OPT *is* **19**× *faster than KeystoneML as measured by cumulative run time over 10 iterations.* By materializing and reusing intermediate results HELIX OPT is able to substantially reduce cumulative run-time relative to other systems. Figure 6(a) shows that 1) on PPR iterations HELIX recomputes only the PPR; 2) the materialization of L/I outputs, which allows the pruning of DPR and L/I in PPR iterations, takes considerably less time than the compute time for DPR and L/I; 3) HELIX OPT reruns DPR in iteration 5 (L/I) because HELIX OPT avoided materializing the large DPR output in a previous iteration. For the first three iterations, which are DPR (the only type of iterations DeepDive supports), the 2× reduction between HELIX OPT and DeepDive is due to the fact that DeepDive does data preprocessing with Python and shell scripts, while HELIX OPT uses Spark. While both KeystoneML and HELIX OPT use Spark, KeystoneML takes longer on DPR and L/I iterations thanHELIX OPT due to a longer L/I time incurred by its caching optimizer's failing to cache the training data for learning.

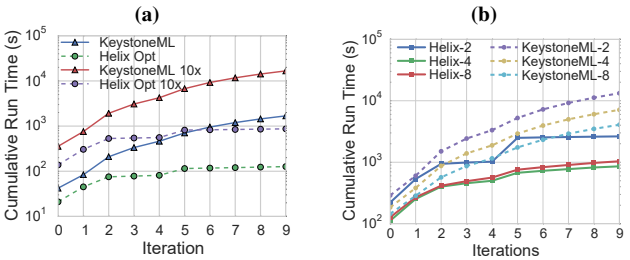

Figure 7: a) Census and Census 10x cumulative run time for HELIX and KeystoneML on a single node; b) Census 10x cumulative run time for HELIX and KeystoneML on different size clusters.

Genomics. In Figure 5(b), HELIX OPT shows a **3**× speedup over KeystoneML for the genomics workflow. The materialize-nothing strategy in KeystoneML clearly leads to no run time reduction in subsequent iterations. HELIX OPT, on the other hand, shows a per-iteration run time that is proportional to the number of operators affected by the change in that iteration. Figure 6(b) shows that 1) in PPR iterations HELIX OPT has near-zero run time, enabled by a small materialization time in the prior iteration; 2) one of the ML models takes considerably more time, and HELIX OPT is able to prune it in iteration 4 since it is not changed.

NLP. Figure 5(c) shows that the cumulative run time for both DeepDive and HELIX OPT increases linearly with iteration for the NLP workflow, but at a much higher rate for DeepDive than HELIX OPT. This is due to the lack of automatic reuse in DeepDive. The first operator in this workflow is a time-consuming NLP parsing operator, whose results are reusable for all subsequent iterations. While both DeepDive and HELIX OPT materialize this operator in the first iteration, DeepDive does not automatically reuse the results. HELIX OPT, on the other hand, consistently prunes this NLP operation in all subsequent iterations, as shown in Figure 6(c), leading to large run time reductions in iterations 1-5 and thus a large cumulative run time reduction.

MNIST. Figure 5(d) shows the cumulative run times for the MNIST workflow. As mentioned above, the MNIST workflow has non-deterministic data preprocessing, which means any changes to the DPR and L/I components prevents safe reuse of any intermediate result. However, iterations containing only PPR changes can reuse intermediates for DPR and L/I had they been materialized previously. Furthermore, we found that the DPR run time is short but cumulative size of all DPR intermediates is large. Thus, materializing all these DPR intermediates would incur a large run time overhead. KeystoneML, which does not materialize any intermediate results, shows a linear increase in cumulative run time due to no reuse. HELIX OPT, on the other hand, only shows slight increase in runtime over KeystoneML for DPR and L/I iterations because it is only materializing the L/I results on these iterations, not the non-reusable, large DPR intermediates. In Figure 6(d), we see 1) DPR operations take negligible time, and HELIX OPT avoids wasteful materialization of their outputs; 2) the materialization time taken in the DPR and L/I iterations pays off for HELIX OPT in PPR iterations, which take negligible run time due to reuse.

6.5.3 *Scalability vs. KeystoneML*

Dataset Size. We test scalability of HELIX and KeystoneML with respect to dataset size by running the ten iterations in Figure 5(a) of the Census Workflow on two different sizes of the input. Census 10x is obtained by replicating Census ten times in order to preserve the learning objective. Figure 7(a) shows run time performance of HELIX and KeystoneML on the two datasets on a single node. Both

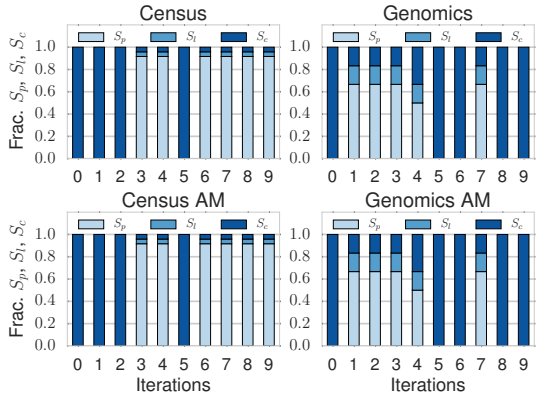

Figure 8: Fraction of states in S_p, S_l, S_c as determined by Algorithm 1 for the Census and Genomics workflows for HELIX OPT and HELIX AM.

yield 10x speedup over the smaller dataset, scaling linearly with input data size, but HELIX continues to dominate KeystoneML.

Cluster. We test scalability of HELIX and KeystoneML with respect to cluster size by running the same ten iterations in Figure 5(a) on Census 10x described above. Using a uniform set of machines, we create clusters with 2, 4, and 8 workers and run HELIX and KeystoneML on each of these clusters to collect cumulative run time.

Figure 7(b) shows that 1) HELIX has lower cumulative run time than KeystoneML on the same cluster size, consistent with the single node results; 2) KeystoneML achieves $\approx 45\%$ run time reduction when the number of workers is doubled, scaling roughly linearly with the number of workers; 3) From 2 to 4 workers, HELIX achieves up to 75% run time reduction 4) From 4 to 8 workers, HELIX sees a slight increase in run time. Recall from Section 3 that the semantic unit data structure in HML allows multiple transformer operations (e.g., indexing, computing discretization boundaries) to be learned using a single pass over the data via loop fusion. This reduces the communication overhead in the cluster setting, hence the super linear speedup in 3). On the other hand, the communication overhead for PPR operations outweighs the benefits of distributed computing, hence the slight increase in 4).

6.6 Evaluation vs. Simpler HELIX Versions

HELIX OPT achieves the lowest cumulative run time on all workflows compared to simpler versions of HELIX. HELIX AM often uses more than 30× the storage of HELIX OPT when able to complete in a reasonable time, while not being able to complete within 50× of the time taken by HELIX OPT elsewhere. HELIX NM takes up to 4× the time taken by HELIX OPT.

Next, we evaluate the effectiveness of Algorithm 2 at approximating the solution to the NP-hard OPT-MAT-PLAN problem. We compare HELIX OPT that runs Algorithm 2 against: HELIX AM that replaces Algorithm 2 with the policy to always materialize every operator, and HELIX NM that never materializes any operator. The two baseline heuristics present two performance extremes: HELIX AM maximizes storage usage, time for materialization, and the likelihood of being able to reuse unchanged results, whereas HELIX NM minimizes all three quantities. HELIX AM provides the most flexible choices for reuse. On the other hand, HELIX NM has no materialization time overhead but also offers no reuse.

Figures 9(a), (b), (e), and (f) show the cumulative run time on the same four workflows as in Figure 5 for the three variants.

HELIX AM *is absent from Figures 9(e) and (f) because it did not complete within* 50× *the time it took for other variants.* The fact that HELIX AM failed to complete for the MNIST and NLP

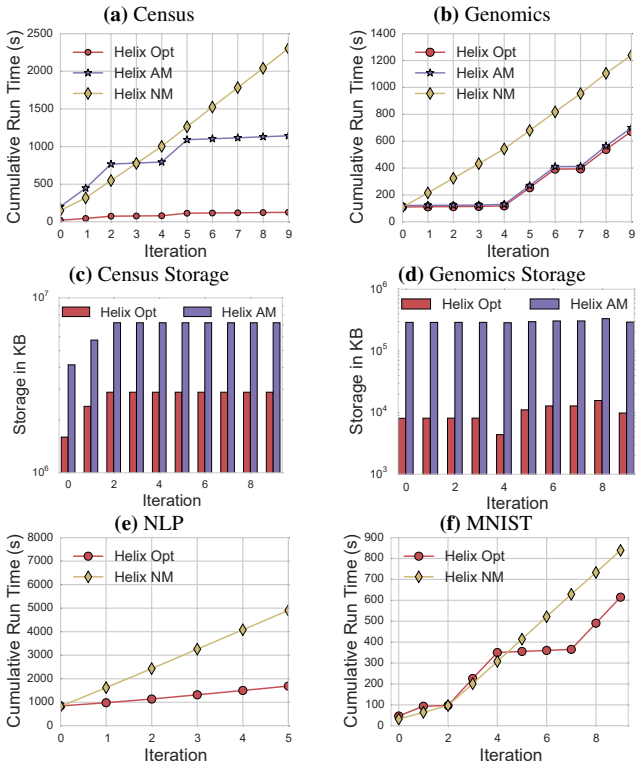

Figure 9: Cumulative run time and storage use against materialization heuristics on the same four workflows as in Figure 5.

workflows demonstrate that indiscriminately materializing all intermediates can cripple performance. Figures 9(e) and (f) show that HELIX OPT achieves substantial run time reduction over HELIX NM using very little materialization time overhead (where the red line is above the yellow line).

For the census and genomics workflows where the materialization time is not prohibitive, Figures 9(a) and (b) show that in terms of cumulative run time, HELIX OPT outperforms HELIX AM, which attains the best reuse as explained above. We also compare the storage usage by HELIX AM and HELIX NM for these two workflows. Figures 9(c) and (d) show the storage size snapshot at the end of each iteration. The x-axis is the iteration numbers, and the y-axis is the amount of storage (in KB) in log scale. The storage use for HELIX NM is omitted from these plots because it is always zero.

We find that HELIX OPT outperforms HELIX AM while using less than half the storage used by HELIX AM for the census workflow in Figure 9(c) and $\frac{1}{30}$ the storage of HELIX AM for the genomics workflow in Figure 9(d). Storage is not monotonic because HELIX purges any previous materialization of original operators prior to execution, and these operators may not be chosen for materialization after execution, thus resulting in a decrease in storage.

Furthermore, to study the optimality of Algorithm 2, we compare the distribution of nodes in the prune, reload, and compute states S_p, S_l, S_c between HELIX OPT and HELIX AM for workflows with HELIX AM completed in reasonable times. Since everything is materialized in HELIX AM, it achieves maximum reuse in the next iteration. Figure 8 shows that HELIX OPT enables the exact same reuse as HELIX AM, demonstrating its effectiveness on real workflows.

Overall, neither HELIX AM *nor* HELIX NM *is the dominant strategy in all scenarios, and both can be suboptimal in some cases.*

7. RELATED WORK

Many systems have been developed in recent years to better support ML workflows. We begin by describing ML systems and other general workflow management tools, followed by systems that target the reuse of intermediate results.

Machine Learning Systems. We describe machine learning systems that support declarative programming, followed by other general-purpose systems that optimize across frameworks.

Declarative Systems. Due to the challenges in developing ML workflows, there has been recent efforts to make it easier to do so declaratively. Boehm et al. categorize declarative ML systems into three groups based on the usage: *declarative ML algorithms*, *ML libraries*, and *declarative ML tasks* [11]. Systems that support *declarative ML algorithms*, such as TensorFlow [4], SystemML [22], OptiML [58], ScalOps [64], and SciDB [57], allow ML experts to program new ML algorithms, by declaratively specifying linear algebra and statistical operations at higher levels of abstraction. Although it also builds a computation graph like HELIX, TensorFlow has no intermediate reuse. TensorFlow's lower level linear algebra operations are not conducive to data preprocessing. HELIX handles reuse at a higher level than TensorFlow ops. *ML libraries*, such as Mahout [44], Weka [25], GraphLab [35], Vowpal Wabbit [33], MLlib [39] and Scikit-learn [48], provide simple interfaces to optimized implementations of popular ML algorithms. TensorFlow has also recently started providing TFLearn [16], a high level ML library targeted at deep learning. Systems that support *declarative ML tasks* allow application developers with limited ML knowledge to develop models using higher-level primitives than in declarative ML algorithms. HELIX falls into this last group of systems, along with DeepDive [75, 17] and KeystoneML [56].

Declarative ML task systems, like HELIX, can seamlessly make use of improvements in ML library implementations, such as MLlib [39], CoreNLP [37] and DeepLearning4j [61], within UDF calls. Unlike declarative ML algorithm systems, that are targeted at ML experts and researchers, these systems focus on end-users of ML.

Systems that Optimize Across Frameworks. These systems target a broad range of use-cases, including ML. Weld [45] and Tupleware [15] optimize UDFs written in different frameworks by compiling them down to a common intermediate representation. Declarative ML task systems like HELIX can take advantage of the optimized UDF implementations; unlike HELIX, these systems do not benefit from seamless specification, execution, and end-to-end optimizations across workflow components that come from a unified programming model.

Systems for Optimizing Data Preprocessing. The database community has identified various opportunities for optimizing DPR. Several approaches identify as a key bottleneck in DPR and optimize it [31, 13, 43, 32]. Another direction aims at reducing the manual effort involved in data cleaning and feature engineering [51, 76, 30, 5, 6]. All of these optimizations are orthogonal to those used by HELIX, which targets end-to-end iterative optimizations.

ML and non-ML Workflow Management Tools. Here we discuss ML workflow systems, production platforms for ML, batch processing workflow systems, and systems for scientific workflows.

ML Workflow Management. Prior tools for managing ML workflows focus primarily on making their pipelines easier to debug. For example, Gestalt [47] and Mistique [62] both tackle the problem of model diagnostics by allowing users to inspect intermediate results. The improved workflow components in these systems could be easily incorporated within HELIX.

ML Platforms-as-Services. A number of industry frameworks [9, 20, 8, 2, 3, 73], attempt to automate typical steps in deploying machine learning by providing a Platform-as-a-Service (PaaS) capturing common use cases. Instead, HELIX is not designed to reduce manual effort of model deployment, but rather model *development*.

General Batch Processing Workflow Systems. A number of companies have implemented workflow management systems for batch processing [10, 59]. These systems are not concerned with runtime optimizations, and rather provide features useful for managing large-scale workflow complexity.

Scientific Workflow Systems. Some systems address the significant mental and computational overhead associated with scientific workflows. VisTrails [12] and Kepler [36] add provenance and other metadata tracking. Other systems attempt to map scientific workflows to cluster resources [71]. One such system, Pegasus [18], also identifies reuse opportunities when executing workflows. The optimization techniques employed by all systems discussed leverage reuse in a simpler manner than does HELIX .

Intermediate Results Reuse. The OEP/OMP problems within HELIX are reminiscent of classical work on view materialization in database systems [14], but operates at a more coarse-grained level on black box operators. COLUMBUS [76] focuses on caching feature columns for feature selection exploration within a single workflow. ReStore [21] manages reuse of intermediates across dataflow programs written in Pig [42], while Nectar [24] does so across DryadLINQ [72] workflows. Jindal et al. [26] study SQL subexpression materialization within a single workflow with many subqueries. In the same vein, Mistique [62] and its spiritual predecessor Sherlock [63] use historical usage as part of their cost models for adaptive materialization. However, the cost models and algorithms proposed in these systems for deciding what to reuse do not consider the operator/subquery dependencies in the DAG and make decisions for each operator independently based on availability, operator type, size, and compute time. We have shown in Figure 4 that decisions can have cascading effects on the rest of the workflow.

8. CONCLUSIONS AND FUTURE WORK

We presented HELIX, a declarative system aimed at accelerating iterative ML application development. In addition to its user friendly, flexible, and succinct programming interface, HELIX tackles two major optimization problems, namely OPT-EXEC-PLAN and OPT-MAT-PLAN, that together enable cross-iteration optimizations resulting in significant run time reduction for future iterations. We devised a PTIME algorithm to solve OPT-EXEC-PLAN by using a reduction to MAX-FLOW. We showed that OPT-MAT-PLAN is NP-HARD and proposed a light-weight, effective heuristic for this purpose. We evaluated HELIX against DeepDive and KeystoneML on workflows from social sciences, NLP, computer vision, and natural sciences that vary greatly in characteristics to test the versatility of our system. We found that HELIX supports a variety of diverse machine learning applications with ease and provides up to a 19× cumulative run time speedup relative to baselines. While HELIX is implemented in a specific way, the techniques and abstractions presented in this work are general-purpose; other systems can enjoy the benefits of HELIX's optimization modules through simple wrappers and connectors.

In future work, we aim to further accelerate iterative workflow development via introspection and querying across workflow versions over time, automating trimming of redundant workflow nodes, as well as auto-suggestion of workflow components to aid workflow development by novices.

9. REFERENCES

[1] Deepdive census example. `https://github.com/HazyResearch/deepdive/tree/master/examples/census`.

[2] Meet michelangelo: Uber's machine learning platform, 2017 (accessed October 8, 2018). https://eng.uber.com/michelangelo/.

[3] *Amazon SageMaker Developer Guide*, 2018 (accessed October 8, 2018). https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[5] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A data system for feature engineering. In *CIDR*, 2013.

[6] M. R. Anderson and M. Cafarella. Input selection for fast feature engineering. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 577–588. IEEE, 2016.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[8] J. Barnes. Azure machine learning microsoft azure essentials, 2015.

[9] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

[10] M. Beauchemin. Airflow: a workflow management platform, 2015 (accessed October 8, 2018). https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8.

[11] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald. Declarative machine learning-a classification of basic properties and types. *arXiv preprint arXiv:1605.05826*, 2016.

[12] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[13] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[14] R. Chirkova, J. Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[15] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[16] A. Damien et al. Tflearn, 2016.

[17] C. De Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang. Deepdive: Declarative knowledge base construction. *SIGMOD Rec.*, 45(1):60–67, June 2016.

[18] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

[19] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.

[20] J. Dunn. Introducing fblearner flow: Facebook's ai backbone, 2018 (accessed October 8, 2018). https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/.

[21] I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[22] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[23] A. D. Gordon. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer, 1995.

[24] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[26] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[27] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[28] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education, 2006.

[29] R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 202–207. AAAI Press, 1996.

[30] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning while learning convex loss models. *arXiv preprint arXiv:1601.03797*, 2016.

[31] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984. ACM, 2015.

[32] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*, pages 19–34. ACM, 2016.

[33] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[35] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[36] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[37] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[38] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[39] X. Meng, J. Bradley, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. 2016.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[41] M. A. Munson. A study on the importance of and time spent on different modeling steps. *ACM SIGKDD Explorations Newsletter*, 13(2):65–71, 2012.

[42] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[43] D. Olteanu and M. Schleich. F: regression models over factorized views. *PVLDB*, 9(13):1573–1576, 2016.

[44] S. Owen, R. Anil, T. Dunning, and E. Friedman. Mahout in action. 2012.

[45] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[46] A. Paszke, S. Gross, S. Chintala, and G. Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

[47] K. Patel, N. Bancroft, S. M. Drucker, J. Fogarty, A. J. Ko, and J. Landay. Gestalt: integrated support for implementation and analysis in machine learning. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 37–46. ACM, 2010.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[49] A. M. Pitts. Operationally-based theories of program equivalence. *Semantics and Logics of Computation*, 14:241, 1997.

[50] W. Rasband. Imagej: Image processing and analysis in java. *Astrophysics Source Code Library*, 2012.

[51] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré. Snorkel: Rapid training data creation with weak supervision. *arXiv preprint arXiv:1711.10160*, 2017.

[52] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[53] X. Ren, J. Shen, M. Qu, X. Wang, Z. Wu, Q. Zhu, M. Jiang, F. Tao, S. Sinha, D. Liem, et al. Life-inet: A structured network-based knowledge exploration and analytics system for life sciences. *Proceedings of ACL 2017, System Demonstrations*, pages 55–60, 2017.

[54] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[55] J. Rosen. Pyspark internals.

[56] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 535–546. IEEE, 2017.

[57] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[58] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[59] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1125–1134. ACM, 2013.

[60] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[61] D. Team. Deeplearning4j: Open-source distributed deep learning for the jvm. *Apache Software Foundation License*, 2, 2016.

[62] M. Vartak, J. M. da Trindade, S. Madden, and M. Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 ACM International Conference on Management of Data*, 2018.

[63] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. In *NIPS Workshop LearningSys*, 2015.

[64] M. Weimer, T. Condie, R. Ramakrishnan, et al. Machine learning in scalops, a higher order cloud computing language. In *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, volume 9, pages 389–396, 2011.

[65] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[66] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19, 2009.

[67] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran. Accelerating human-in-the-loop machine learning: Challenges and opportunities (vision paper). In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18. ACM, 2018.

[68] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran. Helix: Accelerating human-in-the-loop

machine learning (demo paper). *PVLDB*, 11(12):1958–1961, 2018.

[69] D. Xin, L. Ma, S. Song, and A. Parameswaran. How developers iterate on machine learning workflows–a survey of the applied machine learning literature. *KDD IDEA Workshop*, 2018.

[70] D. Xin, S. Macke, L. Ma, R. Ma, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *arXiv preprint arXiv:1812.05762*, 2018.

[71] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[72] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[73] M. Zaharia. Introducing mlflow: an open source machine learning platform, 2018 (accessed October 8, 2018). https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html.

[74] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[75] C. Zhang. *DeepDive: A data management system for automatic knowledge base construction*. PhD thesis, Citeseer, 2015.

[76] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[77] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.