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 reached. Unfortunately, current machine learning systems do not optimize developer-initiated iteration and instead focus on model training—a small fraction of the overall development time. 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 pre-processing, 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 lightweight heuristics for the latter. Empirical evaluation shows that HELIX is not just able to handle a wide variety of use cases in one unified workflow, but is also much faster, providing run time reductions of up to $19 \times$ 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.

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, there have been a number of efforts to make machine learning more declarative, and to speed up the model training process [12]. However, the majority of the development time is in fact spent iterating on the machine learning workflow, incrementally modifying steps within, including the (i) pre-processing steps: by transforming, cleaning, or extracting data differently, or adding, deleting, or transforming features (e.g., via normalization or cleaning); (ii) model training steps: by tweaking parameters, changing the algorithm or regularization; and (iii) post-processing steps: by evaluating the model on test data, or generating statistics or charts. The reason for these iterations is that it is often difficult to predict the performance of a workflow a priori, both due to the variability of the data, and due to 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 the time is actually spent on model training [47], with the bulk of the time spent iterating on all steps of 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 [59]. 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 a word2vec [45]-like approach, and finally cluster the embeddings to find related entities. They repeatedly iterate on this workflow to improve the quality of the relationships discovered. 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 [67], 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 re-run from scratch—often multiple hours on a large cluster 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—many of which are often fairly time-consuming.

One approach to address the expensive recomputation issue is to explicitly materialize all intermediates that do not change across iterations, but this requires writing code to add materialization, identifying what has changed between iterations, and to reuse the materialized results in future iterations. Even if this were a viable option, explicit materialization of all intermediates is extremely wasteful, and figuring out which materialized results to reuse is not straightforward. Since this approach is so cumbersome and wasteful, developers often opt to rerun the entire workflow from scratch.

Unfortunately, existing machine learning systems do not optimize for rapid iteration. For example, KeystoneML [63], 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 [85], materializes the results of all of the feature extraction and engineering steps, while also applying approximate inference to speed up the model training. As we described earlier, while this naive materialization approach does help optimize iterative execution somewhat, this approach 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 $19 \times$ on iterative execution across four real-world applications. By optimizing across iterations, HELIX allows data scientists to avoid wasting time running the entire workflow from scratch every time they make a change.
and instead run machine learning workflows with execution time proportional to the complexity of the change made. HELIX is able to substantially increase the productivity of developers while lowering the consumption of computation resources at the same time.

Developing HELIX involves two types of challenges—in iterative execution optimization and in specification and generalization. **Challenges in Iterative Execution Optimization.** Suppose we can represent a machine learning workflow as a directed acyclic graph, where each node corresponds to a collection of data—be it 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 [85]) is to materialize the result of 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, 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 formally demonstrate that the reuse plan problem is in PTIME via a non-trivial reduction to MAX-FLOW using the PROJECT SELECTION PROBLEM [74], while the materialization problem is, in fact, NP-HARD. Further complicating matters is the fact that the materialization decision needs to be done on the fly because deferring till the end of the execution can cause unnecessary writes/reads from disk due to lack of memory.

**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 pre-processing can vary greatly across applications, often requiring ad hoc code involving complex composition of declarative statements and UDFs [5], making it hard to automatically analyze the workflow to apply holistic iterative execution optimization.

Instead, we adopt a hybrid approach within HELIX: developers write code in an **intuitive and succinct domain-specific language (DSL) based on Scala** (similar to existing machine learning systems like KeystoneML), while also liberally using UDFs as needed to add imperative code, say for feature normalization or extraction. This interoperability allows developers to seamlessly leverage existing machine learning capabilities within Scala [68][57]—thereby retaining all of the benefits of a full imperative language. 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 [77]. HELIX’s DSL not only enables automatic identification of data dependencies and data flow, but also encapsulates all typical machine learning workflow designs. Unlike DeepDive [85], we do not restrict HELIX’s learning paradigm to be regression or factor-graph-based, allowing data scientists to use the most suitable model for their tasks. We demonstrate that all the functions in Scikit-Learn’s (a popular ML toolkit) can be mapped to functions in the DSL. HELIX’s workflows can easily capture supervised or unsupervised learning with applications ranging from natural language processing, to network analysis, to computer vision. **Developing this DSL, while satisfying all of these requirements, was challenging.** 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 for novice developers [76].

**Contributions and Outline.** The rest of the paper is organized as follows: Section 2 presents a quick recap of ML workflows, statistics on how users iterate on ML workflows collected from applied ML literature, an architectural overview of the system and an example workflow; Section 3 describes the programming interface for effortless end-to-end workflow specification; Section 4 discusses the HELIX workflow DAG generation and change tracking; Section 5 presents the solution to the two major optimization problems in accelerating iterative ML. We evaluate our framework on workflows from different domains 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 pre-processing (DPR), where raw data is transformed into ML-compatible representations, learning/inference (LI), where ML models are trained and used to perform inference on new data, and post-processing (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; its DSL also enables embedding of imperative code in declarative statements. Through just a handful of extensible opera-
2.3 Example Workflow

We demonstrate the usage of HELIX with a simple example ML workflow for predicting income using census data from Kohavi [35], shown in Figure 3; this workflow will serve as a running example throughout the paper. Details about the individual operators will be provided in subsequent sections. 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).

Original Workflow: DPR Steps. First, after some variable name declarations, the user defines in line 3-4 a data collection rows read from a data source data consisting of two CSV files, one for training and one for test data, and names the columns of the CSV files age, education, etc. In lines 5-10, the user declares simple features that are values from specific named columns. Note that the user is not required to specify the feature type, which is automatically inferred by HELIX from data. In line 11 ageBucket is declared as a derived feature formed by discretizing age into ten buckets (whose boundaries are computed by HELIX), while line 12 declares an interaction feature, commonly used to capture higher-order patterns, formed out of the concatenation of eduExt and ocExt.

Once the features are declared, the next step, line 13, declares the features to be extracted from and associated with each element of rows. Users do not need to worry about how these features are attached and propagated; users are also free to perform manual feature selection here, studying the impact of various feature combinations, by excluding some of the feature extractors. Finally, as last step of data preprocessing, line 14 declares that an example collection named income to is to be made from rows using target as labels. Importantly, this step converts the features from human-readable formats (e.g., color=red) into an indexed vector representation required for learning.

Original Workflow: L/I & PPR Steps. Line 15 declares an ML model named incPred with type “Logistic Regression” and regularization parameter 0.1, while line 16 specifies that incPred is to be learned on the training data in income and applied on all data in income to produce a new example collection called predictions. Line 17-18 declare a Reducer named checkResults, which outputs a scalar using a UDF for computing prediction accuracy. Line 19 explicitly specifies checkResults’s dependency on target since the content of the UDF is opaque to the optimizer. Line 20 declares that the output scalar named checked is only to be computed from the test data in income. Lines 21 declares that checked must be part of the final output.

Original Workflow: Optimized DAG. The HELIX compiler first translates verbatim the program in Figure 3 into a DAG, which contains all nodes including raceExt and all edges (including the dashed edge) except the ones marked with dots in Figure 3b). This DAG is then transformed by the optimizer, which prunes away raceExt (grayed out) because it does not contribute to the output, and adds the edges marked by dots to link relevant features to the model. DPR involves nodes in purple, and L/I and PPR involve nodes in orange. Nodes with a drum to the right are materialized to disk, either as mandatory output or for aiding in future iterations.

Updated Workflow: Optimized DAG. In the updated version of the workflow, a new feature named masExt is added (below line 9), and clExt is removed (line 13); similarly, in the updated DAG, a new node is added for masExt (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. Similarly, HELIX loads ageBucket instead of recomputing the computing bucket boundaries requiring a full scan. HELIX materializes predictions in both iterations since it has changed. Although predictions is not reused in the updated workflow, its materialization has high expected pay-off over iterations because PPR iterations (changes to checked in this case) are the most common as per our survey results [78].

...
example illustrates that

- Nodes selected for materialization lead to significant speedup in subsequent iterations.
- HELIX reuses results safely, deprecating old results when changes are detected (e.g., predictions is not reused because of the model change).
- HELIX correctly prunes away extraneous operations via dataflow analysis.

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 [44], 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 [61].

3.1 Operations in ML Workflows

Here, we argue that common operations in ML workflows can be decomposed into a small set of basis functions \( \mathcal{F} \). We first introduce these functions and then enumerate its mapping onto operations in Scikit-Learn [54], one of the most comprehensive ML libraries, thereby demonstrating the coverage of \( \mathcal{F} \). In Section 3.2, we introduce HML, which implements capabilities offered by \( \mathcal{F} \).

As mentioned in Section 2, an ML workflow consists of three components: data pre-processing (DPR), learning/inference (L/I), and post-processing (PPR). They are captured by the Transformer, Estimator, and Predictor interfaces in Scikit-Learn, respectively. Similar interfaces can be found in many ML libraries, such as ML-Lib [45], TFX [10], and KeystoneML.

Data Representation. Conventionally, the input space to ML, \( \mathcal{X} \), is a \( d \)-dimensional vector space, \( \mathbb{R}^d \), \( d \geq 1 \), where each dimension corresponds to a feature. Each datapoint is represented by a feature vector (FV), \( x \in \mathbb{R}^d \). For notational convenience, we denote a \( d \)-dimensional FV, \( x \in \mathbb{R}^d \), as \( x^d \). While inputs in some applications can be easily loaded into FVs, e.g., images are 2D matrices that can be flattened into a vector, many others require more complex transformations, e.g., vectorization of text requires tokenization and word indexing. We denote the input dataset of FVs to an ML algorithm as \( D \).

DPR. The goal of DPR is to transform raw input data into \( D \). We use the term record, denoted by \( r \), to refer to a data object in formats incompatible with ML, such as text and JSON, requiring preprocessing. Let \( S := \{ r \} \) be a data source, e.g., a csv file, or a collection of text documents. DPR includes transforming records from one or more data sources from one format to another or into FVs \( \mathbb{R}^d \); as well as feature transformations (from \( \mathbb{R}^d \) to \( \mathbb{R}^d' \)). DPR operations can thus be decomposed into the following categories:

- Parsing \( r \mapsto (r_1, r_2, \ldots) \) : transforming a record into a set of records, e.g., parsing an article into words via tokenization.
- Join \( (r_1, r_2, \ldots) \mapsto r \) : combining multiple records into a single record, where \( r_i \) can come from different data sources.
- Feature Extraction \( r \mapsto x^d \) : extracting features from a record.
- Feature Transformation \( T : x^d \mapsto x'^d \) : deriving a new set of features from the input features.
- Feature Concatenation \( (x^{d_1}, x^{d_2}, \ldots) \mapsto x^{d_1 + d_2} \) : 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 \( x_i \), which is usually estimated empirically by collecting all values of \( x_i \) in \( D \). We address this issue case along with L/I next.

L/I. At a high-level, L/I is about learning a function \( f \) from the input \( D \), where \( f : \mathcal{X} \mapsto \mathbb{R}^{d'} \), \( d' \geq 1 \). This is more general than learning ML models, and also includes feature transformation functions mentioned above. The two main operations in L/I are 1) learning, which produces functions using data from \( D \), and 2) inference, which uses the function obtained from learning to draw conclusions about new data. 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 [33]. Thus, L/I can be decomposed into:

- Learning \( D \mapsto f \) : learning a function \( f \) from the dataset \( D \).
- Inference \( (D, f) \mapsto \hat{Y} \) : using the ML model \( f \) to infer feature values, i.e., labels, \( Y \) from the input FVs in \( D \).

Note that labels can be represented as FVs like other features, hence the usage of a single \( D \) in learning to represent both the training data and labels to unify the abstraction for both supervised and unsupervised learning and to enable easy model composition.

PPR. Finally, a wide variety of operations can take place in PPR, using the learned models and inference results from L/I as input, including model evaluation and data visualization. The most commonly supported PPR operations in general purpose ML libraries are model evaluation and model selection, which can be represented by a computation whose output does not depend on the size of the data \( D \). We refer to a computation with output sizes independent of input sizes as a **reduce**.
Comparison with Scikit-Learn. A dataset in Scikit-Learn is represented as a matrix of FVs, denoted by \( X \). This is conceptually equivalent to \( D = \{ x^i \} \) introduced earlier, as the order of rows in \( X \) is not relevant. Operations in Scikit-Learn are categorized into dataset loading and transformations, learning, and model selection and evaluation [2]. Operations like loading and transformations, learning, and model selection \( X \) is not relevant. Operations in Scikit-Learn are categorized into

dimensionality reduction, and kernel transformations. 

**Scikit-Learn DPR, L/I | Composed Members of \( \mathcal{F} \)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Composed Members of ( \mathcal{F} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit(X, y)</td>
<td>learning (( D \rightarrow f ))</td>
</tr>
<tr>
<td>predict_proba(X)</td>
<td>inference (( f \rightarrow \hat{y} ))</td>
</tr>
<tr>
<td>predict(X)</td>
<td>inference, optionally followed by transformation</td>
</tr>
<tr>
<td>fit_transform(X)</td>
<td>learning, then inference</td>
</tr>
<tr>
<td>transform(X)</td>
<td>transformation or inference, depending on whether operation is learned via prior call to fit</td>
</tr>
</tbody>
</table>

**Scikit-Learn PPR | Composed Members of \( \mathcal{F} \)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Composed Members of ( \mathcal{F} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval: score(f新技术, ytrue)</td>
<td>join ytrue and ypred into a single dataset ( \mathcal{D} ), then reduce</td>
</tr>
<tr>
<td>eval: score(op, X, y)</td>
<td>inference, then, then reduce</td>
</tr>
<tr>
<td>selection: fit(p1, \ldots, pn)</td>
<td>reduce, implemented in terms of learning, inference, and reduce (for scoring)</td>
</tr>
</tbody>
</table>

Table 1: Scikit-Learn DPR, L/I, and PPR coverage in terms of \( \mathcal{F} \).

- **Reduce** (\( \mathcal{D}, s' \) \( \rightarrow \) s): applying an operation on the input dataset \( \mathcal{D} \) and \( s' \), where \( s' \) can be any non-dataset object. For example, \( s' \) can store a set of hyperparameters over which to reduce. 

- **Estimator**, used to indicate that an operation has data-dependent behavior via a \( \text{fit}(X, y) \) method, where \( X \) contains FVs or raw records, and \( y \) contains labels if the operation represents a supervised model.

- **Predictor**, used to indicate that the operation may be used for inference via a \( \text{predict}(X) \) method, taking a matrix of FVs and producing predicted labels. Additionally, if the operation implementing Predictor is a classifier for which inference may produce raw floats (interpreted as probabilities), it may optionally implement \( \text{predict_proba} \).

- **Transformer**, used to indicate that the operation may be used for feature transformations via a \( \text{transform}(X) \) method, taking a matrix of FVs and producing a new matrix \( X_{\text{new}} \).

An operation implementing both Estimator and Predictor has a \( \text{fit}_- \text{-predict} \) method, and an operation implementing both Estimator and Transformer has a \( \text{fit}_- \text{-transform} \) method, for when inference or feature transformation, respectively, is applied immediately after fitting to the data. The rationale for providing a separate Estimator interface is likely due to the fact that it is useful for both feature transformation and inference to have data-dependent behavior determined via the result of a call to \( \text{fit} \). For example, a useful data-dependent feature transformation for a Naïve Bayes classifier maps word tokens to positions in a sparse vector and tracks word counts. The position mapping will depend on the vocabulary represented in the raw training data. Other examples of data-dependent transformations include feature scaling, descretization, imputation, dimensionality reduction, and kernel transformations.

**Coverage in terms of basis functions \( \mathcal{F} \).** The first part of Table 1 summarizes the mapping from Scikit-Learn’s interfaces for DPR and L/I to (compositions of) basis functions from \( \mathcal{F} \). In particular, note that there is nothing special about Scikit-Learn’s use of

Scikit-Learn Operations for PPR. Scikit-Learn interfaces for operations implementing model selection and evaluation are not as standardized as those for DPR and L/I. For evaluation, the typical strategy is to define a simple function that compares model outputs with labels, computing metrics like accuracy or \( F_1 \) score. For model selection, the typical strategy is to define a class that implements methods \( \text{fit} \) and \( \text{score} \). The \( \text{fit} \) method takes a set of hyperparameters over which to search, with different models scored according to the \( \text{score} \) method (with identical interface as for evaluation in Scikit-Learn). The actual model over which hyperparameter search is performed is implemented by an Estimator that is passed into the model selection operation’s constructor.

**Coverage in terms of basis functions \( \mathcal{F} \).** As summarized in the second part of Table 1, Scikit-Learn’s operations for evaluation may be implemented via compositions of (optionally) infer, join, and reduce \( \in \mathcal{F} \). Model selection may be implemented via a reduce that internally uses learning basis functions to learn models for the set of hyperparameters specified by \( s' \), followed by composition with inference and another reduce \( \in \mathcal{F} \) for scoring, eventually returning the final selected model.

### 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 expressions can be found in the technical report [79]. Here, we describe high-level concepts including DCs and operators and discuss the strengths and limitations of HML in Section 3.2.2.

#### 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_E and DC_I 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. Recall that 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 \( D \) 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 \( i \). \( i \) 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. Intuitively, 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 \( S \). In the implementation, the order of SUs in the concatenation is determined globally across \( D \), and SUs whose outputs are not FVs are filtered out.

Sparse vs. Dense Features. The combination of SUs and examples affords ELIXIR 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. Note that users do not have to commit to a single representation for the entire application, since different SUs can contain different types of features. When assembling a mixture of dense and spare FVs, ELIXIR 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 pre-processing steps, eliminating bugs caused by inconsistent data pre-processing procedures handling training and test data separately. ELIXIR 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 ELIXIR are designed to cover the functions enumerated in Section 3.1, using the data structures introduced above. A ELIXIR operator takes one or more DCs and outputs DCs, ML models, or scalars. Each operator enwraps a function \( f \), written in Scala, to be applied to individual elements in the input DCs. As noted above, \( f \) can be learned from the input data or user defined. Like in Scikit-Learn, HML provides off-the-shelf implementations for common operations for ease of use. We describe the relationships between operator interfaces in HML and the function set \( F \) enumerated in Section 3.1 below.

Scanner. Scanner is the interface for parsing \( \in F \) and acts like a flatMap, i.e., for each input element, it adds zero or more elements to the output DC. Thus, it can also be used to perform filtering. The input and output of Scanner are DCs. 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 \( \in F \), 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 (where output FVs from the SUs are automatically assembled into a single FV). In the simple case where each SU in a DC corresponds to an example, a pass-through synthesizer is implicitly declared by naming the output DC, such as in Line 14 of Figure 3b).

Learner. Learner is the interface for learning and inference \( \in F \) in a single operator. A Learner operator \( L \) contains a learned function \( f \), which can be populated by learning from the input data or loading from disk. \( f \) can be an ML model, but it can also be a feature transformation function that needs to be 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. For example, the learner incPred in Line 15 of Figure 3a) is a learner trained on the “train” portion of the DC and outputs inference results as the DC predictions.

Extractor. Extractor is the interface for feature extraction and feature transformation \( \in F \). Extractor contains the function \( f \) applied on the input of SUs, thus the input and output to Extractor are DCs. 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 \( \in F \) and thus the main operator interface for PPR. Like in reduce, the inputs to Reducer are DCs and an optional scalar and the output is a scalar. 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. In Section 3.1 we described how the set of basis operations \( F \) we propose covers all major operations in Scikit-Learn, one of the most comprehensive ML libraries. We then showed in Section 3.2 that HML captures all functions in \( F \).

However, we must note that HML is not nearly as mature as Scikit-Learn. 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 and Weka, 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. For example, since all ML-Lib models implement the train (equivalent to learning) and predict (equivalent to inference) methods, they can easily be plugged into Learner in ELIXIR. We demonstrate in Section 6 that the current set of operations is sufficient for supporting diverse applications.

Limitations. Since ELIXIR currently relies on its Scala DSL for workflow specification, popular non-JVM libraries, such as TensorFlow and Pytorch, cannot be imported easily without significantly degrading performance compared to their native runtime environment. Developers with workflows implemented in other languages will need to translate them into HML, which should be straightforward due to the natural correspondence between ELIXIR operators and those in standard ML libraries. That said, our contributions in materialization and reuse are language-independent and apply across all languages. In the future, we plan on abstracting the DAG representation in ELIXIR into a language-agnostic system that can sit below the language layer for all DAG based systems, including TensorFlow, Scikit-Learn, Spark.

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

4. COMPILED AND REPRESENTATION

In this section, we describe the Workflow DAG, the abstract model used internally by ELIXIR 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, ELIXIR’s intermediate code generator constructs a Workflow DAG from HML declarations, with nodes corresponding to operator outputs, (DCs, scalars, or ML models), and edges corresponding to input-output relationships between operators.

Definition 1. For a Workflow \( W \) containing ELIXIR operators \( F = \{f_1\} \), the Workflow DAG is a directed acyclic graph \( G_W = \)
(N, E), where node n_t ∈ N represents the output of f_t ∈ F and (n_{t−1}, n_t) ∈ E if the output of f_t is an input to f_{t+1}.

Figure 3 shows the Workflow DAG for the program in Figure 3a. Nodes for operators involved in DPR are colored purple whereas those involved in I/O and PPR are colored orange. 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 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 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 ≥ 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^t_i ∈ N_t is equivalent to n^t+1_i ∈ N_{t+1}, denoted as n^t_i ≈ n^t+1_i + 1, if a) the operators corresponding to n^t_i and n^t+1_i compute identical results on the same inputs and b) n^t_i ≡ n^t+1_i ∨ n^t_i ∈ parents(n^t_i), n^t+1_i ∈ parents(n^t+1_i). We say n^t+1_i ∈ N_{t+1} is original if it has no equivalent node in N_t.

Equivalence is symmetric, i.e., n^t_i ≈ n^t+1_i ⇔ n^t+1_i ≈ n^t_i, and transitive, i.e., (n^t_i ≈ n^t+1_i ∧ n^t+1_i ≈ n^t+2_i) ⇒ n^t_i ≈ n^t+2_i. 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^t_i ∈ N_t has an equivalent materialization if n^t_i is stored on disk, where t ≤ t and n^t_i ≈ n^t_i.

One challenge in determining equivalence is deciding whether two versions of an operator compute the same results on the same input, which is undecided in general as proven by Rice’s Theorem [60]. There has been a lot of work on verifying operational equivalence for specific classes of programs [75][56][27]. 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. We plan to incorporate more advanced techniques for verifying equivalence as future work. For result correctness, i.e., that results obtained at iteration t reflect the specification for W_t, we impose the constraint:

Constraint 1. At iteration t + 1, if an operator n^t+1_i is original, it must be recomputed.

With Constraint 1 our current approach to tracking changes yields the following guarantee on result correctness:

Theorem 1. HELIX returns correct results if the changes between iterations are made only within the programming interface, i.e., all other factors, such as library versions and files on disk, stay invariant, i.e., unchanged, between executions at iteration t and t + 1.

Proof. First, note that the results for W_0 are correct since there is no reuse at iteration 0. Suppose for contradiction that the results at t are correct, the results at iteration t + 1 are incorrect, i.e., ∃ n^t+1_i s.t. the results for n^t_i are reused when n^t+1_i is original. Under the invariant conditions in Theorem 1 we can only have n^t+1_i ≠ n^t_i if the code for n_i changed or the code changed for an ancestor of n_i. Since HELIX detects all code changes, it identifies all original operators. Thus, for the results to be incorrect in HELIX, we must have reused n^t_i for some original n^t+1_i. However, this violates Constraint 1. Therefore, the results for W_t are correct ∀ t ≥ 0.

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 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 ∈ N corresponding to the output of the operator f_i is associated 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. In a subsequent iteration, n_i can be loaded back 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 = ∞. Root nodes in the Workflow DAG, which correspond to data sources, have l_i = c_i.

Operator State. In the execution of workflow W, each node n_i typically 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) ∈ {S_l, S_c, S_p} denote the state of each n_i ∈ 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 ∈ N, if s(n_i) = S_c then s(n_i) ≠ S_p for every n_j ∈ parents(n_i).

Workflow Run Time. A node n_i in state S_l, S_c, 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

\[ T(W, s) = \sum_{n_i ∈ N} I\{s(n_i) = S_l\} c_i + I\{s(n_i) = S_c\} l_i \]  

(1)

where I\{ \} is the indicator function.

Clearly, setting all nodes to S_l 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) ≠ S_p ∀ n_j ∈ parents(n_i). Deciding whether to load or compute the parents can have a cascading effect on the states of their ancestors. We explore how to determine the states for each nodes to minimize Equation 1 next.

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.

7
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 node $n_i \in N$, 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 Constraints 1 and Constraint 2.

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

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

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 $n_i$ with equivalent materialization needs to be recomputed (Constraint 1).

Choosing 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 1 disallows the parents of computed nodes to be pruned. Thus, the decision 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 $n_{10-6}$ to be potentially pruned. However, the decision to compute $n_8$, possibly arising from the fact that $n_8 \iff c_8$, requires that $n_7$ 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 [14].

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 to an equivalent instance of PSP as follows: Fix an optimal solution to $y$ maps to an optimal solution of $x$. Let $G = (N,E)$ 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, $\phi$, 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 $b_j$ 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 $a_4$, $a_5$, $a_6$, and $\gamma$, $\delta$, $\epsilon$, $\zeta$, $\theta$, $\alpha$, $\beta$, and $\gamma$ are selected, which cause nodes $n_4$, $n_5$, $n_6$, to be loaded, $n_7$, and $n_7$ to be computed, and $n_8$, $n_9$, $n_{10}$, 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 $l_i - c_i$. 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 1. 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 (i.e., $l_j = \infty$) 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 2. Given an instance of OPT-EXEC-PLAN $x$, the reduction in Algorithm 1 produces a feasible and optimal solution to $x$.

Proof. See Appendix B.

The computational complexity of OEP is $O(N \cdot E)$, while the complexity of PSP is $O(P \cdot P)$. Thus, the overall complexity is $O(N \cdot E + P \cdot P)$. The Edmonds-Karp algorithm [23] for MAX-FLOW, which runs in time $O(N \cdot E \cdot |E|^2)$, is used to solve the PSP instance.

The impact of change detection precision and recall. The optimality of our algorithm for OEP assumes that the changes between iteration $t$ and $t+1$ have been identified perfectly. In reality, this may not be the case due to the intractability of change detection, as discussed in Section 4.3. An undetected change is a false negative in this case, while falsely identifying an unchanged operator as deprecated is a false positive. A detection mechanism with high precision lowers the chance of unnecessary recomputation, whereas anything lower than perfect recall leads to incorrect results. In our current
approach, we opt for a detection mechanism that guarantee correctness under mild assumptions, at the cost of some false positives such as $a + b \neq b + a$.

### 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 $t > t$. 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 5.2), 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: $l_i = s_i/(disk\ read\ speed)$. 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 iterations, $T$, is unknown. Since modeling $T$ is a complex open problem, we defer the amortization model to future work.

**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

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

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, next.

**Problem 3.** (OPT-MAT-PLAN) Given a Workflow $W_t$ with DAG $G_W = (N_t, E_t)$ 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 $\sum_{n_i \in M} s_i \leq S$.

Let $M^*$ be the optimal solution to OMP, i.e.,

$$\arg\min_{M \subseteq N_t} \sum_{n_i \in M} l_i + T^*(W_{t+1})$$

As discussed in [78], there are many possibilities for $W_{t+1}$, and they vary by application domain. User modeling and predicative 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 3.** OPT-MAT-PLAN is NP-hard.

**Proof.** See Appendix C.

**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, after a definition.

**Definition 5.** Given a Workflow $W_t$ with 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.

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

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

where $t(n_i) = \mathbb{I}\{s(n_i) = S_j\} c_i + \mathbb{I}\{s(n_i) = S_k\} l_i$.

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.

Note that the decision to materialize does not depend on which ancestor nodes have been previously materialized. The advantage of this approach is that regardless of where in the workflow the changes are made, the reusable portions leading up to the changes are likely to have an efficient execution plan. That is to say, if it is cheaper to load a reusable node $n_i$ than to recompute, Algorithm 2 would have materialized $n_i$ previously, allowing us to make the right choice for $n_i$. Otherwise, Algorithm 2 would have materialized some ancestor $n_j$ of $n_i$ such that loading $n_j$ and computing everything leading to $n_i$ is still cheaper than loading $n_i$.

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 are ignored by Algorithm 2—we cannot retroactively update our decision for $n_5$ after $n_8$ has been run. We show in Section 5.2 that this simple algorithm is effective in multiple application domains.

**Limitations of Streaming OMP.** The streaming OMP heuristic given in Algorithm 2 can behave poorly in pathological cases. For one simple example, consider a workflow given by a chain DAG of $m$ nodes, where node $n_1$ (starting from $i = 1$) is a prerequisite for node $n_{i+1}$. If node $n_i$ has $l_i = i$ and $c_i = 3$, for all $i$, then Algorithm 2 will choose to materialize every node, which has storage costs of $O(m^2)$, whereas a smarter approach would only materialize later nodes and perhaps have storage cost $O(m)$.
Data-driven pruning is a powerful technique that can be extended to unlock the possibilities for many more impactful automatic workflow optimizations. Possible future work includes using this technique to minimize online inference time in large scale, high query-per-second settings and to adapt the workflow online in stream processing.

**Cache Pruning.** While Spark, the underlying data processing engine for **HELIX**, provides automatic data uncaching via a least-recently-used (LRU) scheme, **HELIX** improves upon the performance by actively managing the set of data to evict from cache.

From the DAG, **HELIX** can detect when a node becomes out-of-scope. Once an operator has finished running, **HELIX** analyzes the DAG to uncache newly out-of-scope nodes. Combined with the lazy evaluation order, the intermediate results for an operator reside in cache only when it is immediately needed for a dependent operator.

One limitation of this eager eviction scheme is that any dependencies undetected by **HELIX**, such as the ones created in a UDF, can lead to premature uncaching of DCs before they are truly out-of-scope. The uses keyword in HML, described in Table 5, provides a mechanism for users to manually prevent this by explicitly declaring a UDF’s dependencies on other operators. In the future, we plan on providing automatic UDF dependency detection via introspection.
<table>
<thead>
<tr>
<th>Num. Data Source</th>
<th>Census (Source: [1])</th>
<th>Genomics (Source: [59])</th>
<th>IE (Source: [19])</th>
<th>MNIST (Source: [23])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input to Example Mapping</td>
<td>One-to-One</td>
<td>One-to-Many</td>
<td>One-to-Many</td>
<td>One-to-One</td>
</tr>
<tr>
<td>Feature Graminality</td>
<td>Fine Grained</td>
<td>N/A</td>
<td>Fine Grained</td>
<td>Coarse Grained</td>
</tr>
<tr>
<td>Learning Task Type</td>
<td>Supervised; Classification</td>
<td>Unsupervised</td>
<td>Structured Prediction</td>
<td>Supervised; Classification</td>
</tr>
<tr>
<td>Application Domain</td>
<td>Social Sciences</td>
<td>Natural Sciences</td>
<td>NLP</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>Supported by HELIX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Supported by DeepDive</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

Table 2: 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.

- **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 pre-processing 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 [21], 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. This workflow evaluates a system’s efficiency in handling simple ML tasks with fine-grained feature engineering.

**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 [46]; 2) cluster the vector representation of genes using K-Means to identify functional similarity. Each input record is an article, and it maps onto many gene names, which are training examples. This workflow has minimal data pre-processing with specific features but involves multiple learning steps. Both learning steps are unsupervised, which leads to more informative and exploratory evaluations of the model outputs than the standard metrics used for supervised learning. 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 [19]. The objective is to extract structured information from unstructured input text, using complex fine-grained features such as part-of-speech tagging. Each input article contains ≥ 0 spouse pairs, hence creating a one-to-many relationship between input records and learning examples. This workflow exemplifies use cases in natural language processing (NLP) and information extraction, and tests a system’s ability to handle joins and complex data pre-processing.

**MNIST Workflow.** The MNIST dataset [40] contains images of handwritten digits to be classified, which is a well-studied task in the computer vision community, from the KeystoneML [63] evaluation. The workflow involves nondeterministic (and hence not reusable) data pre-processing, 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 2 indicating that we used an existing implementation by the developers of DeepDive or KeystoneML, which can be found at:

- **Census DeepDive**: [https://github.com/HazyResearch/deepdive/blob/master/examples/census/app.ddlog](https://github.com/HazyResearch/deepdive/blob/master/examples/census/app.ddlog)

DeepDive has its own DSL, while KeystoneML’s programming interface is embedded 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 [78] 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), double the average iteration count found in our survey [78].

Note that in real world use, the modifications in each iteration are entirely up to the user. HELIX is not designed to suggest modifications, and the modifications chosen in our experiments are for evaluating only system run time and storage use. We use statistics aggregated over > 100 papers to determine the iterative modifications in order to simulate behaviors of the average domain expert more realistically than arbitrary choice.

**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. We use Postgres as the database backend for DeepDive. 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. The cumulative run time considers only the run time of the workflows, not any human development time. We measure with wall-clock time because it is the latency experienced by the user. 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 to analyze the effect of batch processing, 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 2 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 pre-processing. On the other hand, DeepDive is highly specialized for information extraction and focuses on supporting data pre-processing. 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 $19 \times$ 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 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 $19 \times$ 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 returns 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 $2 \times$ reduction between HELIX OPT and DeepDive is due to the fact that DeepDive does data pre-processing 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 than does HELIX OPT because its caching optimizer does not cache prepared training data. The dominant number of PPR iterations for this workflow reflects the fact that users in the social sciences conduct extensive fine-grained analysis of results, per our literature survey [78].

Genomics. In Figure 5(b) HELIX OPT shows a $3 \times$ 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 5(c) leading to large run time reductions in iterations 1-5. HELIX OPT reruns the changing portion of DPR in iterations 1-5.

MNIST. Figure 5(d) shows the cumulative run times for the MNIST workflow. As mentioned above, the MNIST workflow has non-deterministic data pre-processing, 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 unreusable, large DPR intermediates. In Figure 5(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. Census10x 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. Both 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(c) 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 ≈ 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 for 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 8(a), (b), (c) and (d) show the cumulative run time on the same four workflows as in Figure 5 for the three variants.

HELIX AM is absent from Figures 8(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 workflows demonstrate that indiscriminately materializing all intermediates can cripple performance. Figures 8(c) and 8(d) 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 8(a) and 8(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 8(c) and 8(d) show the 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 8(c) and 8(d), the storage of HELIX AM for the genomics workflow in Figure 8(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 choices made by Algorithm, we compare the distribution of nodes in the prune, reload, and compute states $S_p, S_r, S_c$ between HELIX OPT and HELIX AM on workflows for which HELIX AM completed in reasonable times. Since HELIX AM always materializes intermediates, it achieves maximum reuse in the next iteration. Figure 8(c) shows that HELIX OPT enables the same reuse as does 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.

6.7 Memory Usage by HELIX

We evaluate memory usage by HELIX to ensure that its materialization and reuse benefits do not come at the expense of large memory overhead. We measure memory usage at one-second intervals during HELIX workflow execution. Figure 10 shows the peak and average memory used by HELIX in each iteration for all four workflows. We allocate 30GB memory (25% of total available memory) in the experiments. We observe that HELIX runs within the memory constraints on all workflows. Furthermore, on iterations where HELIX reuses intermediate results to achieve a high reduction in run time compared to other systems, memory usage is also significantly reduced. This indicates that HELIX reuses small intermediates that enable the pruning of a large portion of the subgraph to reduce run time, instead of overloading memory.

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 [13]. Systems that support declarative ML algorithms, such as TensorFlow [5], SystemML [26], and OptiML [63]. These systems perform workflow-level optimizations to reduce end-to-end execution time. Finally, at the extreme end of this spectrum are systems for in-RDBMS analytics [59, 65] that extend databases to support ML, at great cost to flexibility.

Declarative ML task systems, like HELIX, can seamlessly make use of improvements in ML library implementations, such as MLlib [53], CoreNLP [43], and DeepLearning4J [68], 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-related tasks and Tupleware [17] 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 Pre-Processing.** The database community has identified various opportunities for optimizing DPR. Several approaches identify as a key bottleneck in DPR, and optimize it [37, 14, 39, 88]. Kumar et al. [37] optimizes generalized linear models directly over factorized / normalized representations of relational data, avoiding key-foreign key joins. Morpheus [15] and F [49] extend this factorized approach to general linear algebra operations and linear regression models, respectively (the latter over arbitrary joins). Some work [33] even attempts to characterize when joins can be eschewed altogether, without sacrificing performance. All of these optimizations are orthogonal to those used by Helix. Another direction aims at reducing the manual effort involved during data cleaning and feature engineering [38, 86, 85, 6, 22]. All of these optimizations are orthogonal to the end-to-end iterative optimizations used by Helix. Snorkel [88] supports training data engineering using rules. Columbus [86] optimizes feature selection specifically for regression models. ActiveClean [80] integrates data cleaning with learning convex models, using gradient-biased samples to identify dirty data. Brainwash [6] proposes to expedite feature engineering by recommending feature transformations. Zombie [7] speeds up data preparation by learning over smaller, actively-learned informative subsets of data during feature engineering. These approaches are bespoke for the data pre-processing portion of ML workflows and do not target end-to-end optimizations, although there is no reason they could not be integrated within Helix.

**ML and non-ML Workflow Management Tools.** Here we discuss ML workflow systems, production platforms for ML, industry 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 [33] and Mistique [70] 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 [10, 22, 9, 8, 33] attempt to automate typical steps in deploying machine learning by providing a Platform-as-a-Service (PaaS) capturing common use cases. These systems vary in generality — frameworks like SageMaker, Azure Studio, and MLFlow are built around services provided by Amazon, Microsoft, and DataBricks, respectively, and provide general solutions for production deployment of ML models for companies that in-house infrastructure. On the other hand, TFX, FBLearner Flow, and Michelangelo are optimized for internal use at Google, Facebook, and Uber, respectively. For example, TFX is optimized for use with TensorFlow, and Michelangelo is optimized for Uber’s real-time requirements, allowing production models to use features extracted from streams of live data.

The underlying “workflow” these frameworks manage is not always given an explicit representation, but the common unifying thread is the automation of production deployment, monitoring, and continuous retraining steps, thereby alleviating engineers from the labor of ad-hoc solutions. Helix is not designed to reduce manual effort of model deployment, but rather model development. The workflows managed by Helix sit at a lower level than those of industry PaaS systems and therefore require different techniques.

**General Batch Processing Workflow Systems.** A number of companies have implemented workflow management systems for batch processing [11, 69]. 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 [14] and Kepler [42] add provenance and other metadata tracking to visualization-producing workflows, allowing for reproducibility, easier visualization comparison, and faster iteration. Other systems attempt to map scientific workflows to cluster resources [81]. One such system, Pegasus [20], also identifies reuse opportunities when executing workflows. The optimization techniques employed by all systems discussed leverage reuse in simpler manners. Helix, since the workflows are coarser-grained and computation-heavy, so that the cost of loading cached intermediate results can be considered negligible.

**Intermediate Results Reuse.** The OEP/OMP problems within Helix are reminiscent of classical work on view materialization in database systems [16], but operates at a more coarse-grained level on black box operators. However, the reuse of intermediate results within ML workflows differs from traditional database view materialization in that it is less concerned with fine-grained updates, and instead treats operator outputs as immutable black-box units due to the complexity of the data analytics operator. Including Intermediate result reuse has been studied in various contexts. Columbus [86] focuses on caching feature columns for feature selection exploration within a single workflow. ReStore [24] manages reuse of intermediates across dataflow programs written in Pig [48], while Nectar [28] does so across DryadLINQ [82] workflows. Jindal et al. [32] study SQL subexpression materialization within a single workflow with many subqueries. Perez et al. [55] also study SQL subexpression materialization, but in an inter-query fashion that uses historical data to determine utility of materialization for future reuse. In the same vein, Mistique [70] and its spiritual predecessor Sherlock [71] use historical usage as part of their cost models for adaptive materialization. Some workflow systems already discussed leverage simplified caching. Helix shares some similarities with the systems above but also differs in significant ways. Mistique [70], Nectar [28], and ReStore [24] share the goal of efficiently reusing ML workflow intermediates with Helix. However, the cost models and algorithms proposed in these systems for deciding what to reuse pay little attention to 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. The reuse problems studied in Columbus [86] and Jindal et al. [32] differ from ours in that they are concerned with decomposing a set of queries $Q$ into subqueries and picking the minimum cost set of subqueries to cover $Q$. The queries and subqueries can be viewed as a bipartite graph, and the optimization problem can be cast as a SET COVER. They do not handle iteration but rather efficient execution of parallel queries. Furthermore, the algorithms for choosing what to materialize in Mistique [70] and Perez et al. [55] use historical data as signals for likelihood of reuse in the future, whereas our algorithm directly uses projected savings for the next iteration based on the reuse plan algorithm. Their approaches are reactive, while ours is proactive.

## 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 40-60% cumulative run time reduction on complex learning tasks and nearly an order of magnitude reduction on simpler ML tasks compared to both DeepDive and KeystoneML. 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. Specifically, HELIX is capable of tracing specific features in the ML model to the operators in the DAG. This allows information about feature importance learned in the ML model to be used directly to prune the DAG. In addition, the materialization and reuse techniques we proposed can be extended to optimize parallel executions of similar workflows.
9. REFERENCES


APPENDIX

A. HML SPECIFICATIONS

\[
\begin{align*}
\text{(var)} & ::= (\text{string}) \\
\text{(scanner)} & ::= (\text{var})| (\text{scanner-obj}) \\
\text{(extractor)} & ::= (\text{var})| (\text{extractor-obj}) \\
\text{(typed-ext)} & ::= 'C' (\text{var}) ', ' (\text{extractor}) ')' \\
\text{(extractors)} & ::= 'C' (\text{extractor}) { ',', (\text{extractor}) } ')' \\
\text{(typed-exs)} & ::= 'C' (\text{typed-ext}) { ',', (\text{typed-ext}) } ')' \\
\text{(obj)} & ::= (\text{data-source}) | (\text{scanner}) | (\text{extractor-obj}) | (\text{learner-obj}) | (\text{synthesizer-obj}) | (\text{reducer-obj}) \\
\text{(assign)} & ::= (\text{var}) 'refers_to' (\text{obj}) \\
\text{(expr1)} & ::= (\text{var}) 'is_read_into' (\text{var}) 'using' (\text{scanner}) \\
\text{(expr2)} & ::= (\text{var}) 'has_extractors' (\text{extractors}) \\
\text{(list)} & ::= (\text{var}) | 'C' (\text{var}) ', ' (\text{var}) | 'C' (\text{var}) ')' \\
\text{(apply)} & ::= (\text{var}) 'on' (\text{list}) \\
\text{(expr3)} & ::= (\text{apply}) 'as_examples' (\text{var}) \\
\text{(expr4)} & ::= (\text{apply}) 'as_results' (\text{var}) \\
\text{(expr5)} & ::= (\text{var}) 'as_examples' (\text{var}) 'with_labels' (\text{extractor}) \\
\text{(expr6)} & ::= (\text{var}) 'uses' (\text{typed-exs}) \\
\text{(expr7)} & ::= (\text{var}) 'is_output' () \\
\text{(statement)} & ::= (\text{assign}) | (\text{expr1}) | (\text{expr2}) | (\text{expr3}) | (\text{expr4}) | (\text{expr5}) | (\text{expr6}) | (\text{expr7}) | (\text{Scala expr}) \\
\text{(program)} & ::= 'object' (\text{string}) 'extends Workflow' '{ (\text{statement}) | (\text{break}) | (\text{line}) }' \\
\end{align*}
\]

Figure 11: HELIX syntax in Extended Backus-Naur Form. <string> denotes a legal String object in Scala; <*-obj> denotes the correct syntax for instantiating object of type "*-"; <Scala expr> denotes any legal Scala expression. A HELIX Workflow can be comprised of any combination of HELIX and Scala expressions, a direct benefit of being an embedded DSL.

B. PROOF OF THEOREM 2

For clarity, we first formulate OPT-EXEC-PLAN as an integer linear program before presenting the proof itself.

**Integer Linear Programming Formulation.** Problem 2 can be formulated as an integer linear program (ILP) as follows. First, for each node \( n_i \in G \), introduce binary indicator variables \( X_{a_i} \) and \( X_{b_i} \), defined as follows:

\[
\begin{align*}
X_{a_i} &= \{ s(n_i) \neq S_p \} \\
X_{b_i} &= \{ s(n_i) = S_p \}
\end{align*}
\]

That is, \( X_{a_i} = 1 \) if node \( n_i \) is not pruned, and \( X_{b_i} = 1 \) if node \( n_i \) is computed. Note that it is not possible to have \( X_{a_i} = 0 \) and \( X_{b_i} = 1 \). Also note that these variables uniquely determine node \( n_i \)'s state \( s(n_i) \).

With the \( \{X_{a_i}\} \) and \( \{X_{b_i}\} \) thus defined, our ILP is as follows:

\[
\begin{align}
\text{minimize} & \quad \sum_{i=1}^{N} X_{a_i} c_i + X_{b_i} (c_i - l_i) \\
\text{subject to} & \quad X_{a_i} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \tag{6a} \\
& \quad \sum_{n_j \in P(n_i)} X_{a_j} - X_{b_j} \geq 0, \quad 1 \leq i \leq |N|, \tag{6b} \\
& \quad X_{a_i}, X_{b_i} \in \{0, 1\}, \quad 1 \leq i \leq |N| \tag{6c}
\end{align}
\]

Equation (6a) prevents the assignment \( X_{a_i} = 0 \) (\( n_i \) is pruned) and \( X_{b_i} = 1 \) (\( n_i \) is computed), since a pruned node cannot also be computed by definition. Equation (6c) is equivalent to Constraint 1 if \( X_{b_i} = 1 \) (\( n_i \) is computed), any parent \( n_j \) of \( n_i \) must not be pruned, i.e., \( X_{a_j} = 1 \), in order for the sum to be nonnegative. Equation (6b) requires the solution to be integers.

This formulation does not specify a constraint corresponding to Constraint 1. Instead, we enforce Constraint 1 by setting the load and compute costs of nodes that need to be recomputed to specific values, as inputs to Problem 2. Specifically, we set the load cost to \( \infty \) and the compute cost to \( -\epsilon \) for a small \( \epsilon > 0 \). With these values, the cost of a node in \( S_i, S_p, S_c \) is \( \infty, 0, -\epsilon \) respectively, which makes \( S_c \) a clear choice for minimizing Eq (6a).

Although ILPs are, in general, NP-Hard, the astute reader may notice that the constraint matrix associated with the above optimization problem is totally unimodular (TU), which means that an optimal solution for the LP-relaxation (which removes constraint 6c in the problem above) assigns integral values to \( \{X_{a_i}\} \) and \( \{X_{b_i}\} \), indicating that it is both optimal and feasible for the problem above as well.

In fact, it turns out that the above problem is the dual of a flow problem; specifically, it is a minimum cut problem. This motivates the reduction introduced in Section 5.2.

**Main proof.** The proof for Theorem 2 follows directly from the two lemmas proven below. Recall that given an optimal solution \( A \) to PSP, we obtain the optimal state assignments for OEP using the following mapping:

\[
s(n_i) = \begin{cases} 
S_c & \text{if } a_i \in A \text{ and } b_i \in A \\
S_i & \text{if } a_i \in A \text{ and } b_i \notin A \\
S_p & \text{if } a_i \notin A \text{ and } b_i \notin A 
\end{cases} \tag{7}
\]

**Lemma 1.** A feasible solution to PSP under \( \varphi \) also produces a feasible solution to OEP.

**Proof.** We first show that satisfying the prerequisite constraint in PSP leads to satisfying Constraint 2 in OPT-EXEC-PLAN. Suppose for contradiction that a feasible solution to PSP under \( \varphi \) does not produce a feasible solution to OEP. This implies that for some node \( n_i \in N \) s.t. \( s(n_i) = S_c \), at least one parent \( n_j \) has \( s(n_j) = S_p \). By the inverse of Eq (7), \( s(n_j) = S_i \) implies that \( b_i \) was selected, while \( s(n_j) = S_p \) implies that neither \( a_j \) nor \( b_j \) was selected. By construction, there exists an edge \( a_j \rightarrow b_i \). The project selection entailed by the operator states leads to a violation of the prerequisite constraint. Thus, a feasible solution to PSP must produce a feasible solution to OEP under \( \varphi \).

**Lemma 2.** An optimal solution to PSP is also an optimal solution to OEP under \( \varphi \).

**Proof.** Let \( Y_{a_i} \) be the indicator for whether project \( a_i \) is selected, \( Y_{b_i} \) for the indicator for \( b_i \), and \( p(x_i) \) be the profit for project \( x_i \). The optimization object for PSP can then be written as

\[
\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{N} Y_{a_i} p(a_i) + Y_{b_i} p(b_i) \tag{8}
\]

Substituting our choice for \( p(a_i) \) and \( p(b_i) \), Eq (8) becomes

\[
\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{N} -Y_{a_i} l_i + Y_{b_i} (l_i - c_i) \tag{9}
\]

The mapping established by Eq (7) is equivalent to setting \( X_{a_i} = Y_{a_i} \) and \( X_{b_i} = Y_{b_i} \). Thus the maximization problem in Eq (9) is equivalent to the minimization problem in Eq (6a), and we obtain an optimal solution to OEP from the optimal solution to PSP.

20
Under this assumption, we achieve maximum reusability of materialized intermediate results since all operators that persist across iterations $t$ and $t + 1$ are equivalent. We use this assumption to sidestep the problem of predicting iterative modifications, which is a major open problem by itself.

Our proof for the NP-hardness of OMP subject to Eq. (11) uses a reduction from the known NP-hard Knapsack problem.

**Problem 4. (Knapsack)** Given a knapsack capacity $B$ and a set $N$ of $n$ items, with each $i \in N$ having a size $s_i$ and a profit $p_i$, find 

$$S^* = \arg\max_{S \subseteq N} \sum_{i \in S} p_i$$

such that $\sum_{i \in S} s_i \leq B$.

For an instance of Knapsack, we construct a simple Workflow DAG $W$ as shown in Figure 12. For each item $i$ in Knapsack, we construct an output node $n_i$ with $l_i = s_i$ and $c_i = p_i + 2s_i$. We add an input node $n_0$ with $l_0 = \epsilon \leq \min_i s_i$ that all output nodes depend on. Let $Y_i \in \{0, 1\}$ indicate whether a node $n_i \in M$ in the optimal solution to OMP in Eq. (4) and $X_i \in \{0, 1\}$ indicate whether an item is picked in the Knapsack problem. We use $B$ as the storage budget, i.e., $\sum_{i \in \{0, 1\}} Y_i l_i \leq B$.

**Theorem 4.** We obtain an optimal solution to the Knapsack problem for $X_i = Y_i \ \forall i \in \{1, 2, \ldots, n\}$.

**Proof.** First, we observe that for each $n_i$, $T^*(W)$ will pick $\min(l_i, c_i)$ given the flat structure of the DAG. By construction, $\min(l_i, c_i) = l_i$ in our reduction. Second, materializing $n_i$ helps in the first iteration only when it is loaded in the second iteration. Thus, we can rewrite Eq. (4) as 

$$\arg\min_{Y \in \{0, 1\}^N} \sum_{i=1}^N Y_i l_i + \sum_{i=1}^N Y_i l_i + (1 - Y_i) c_i$$

where $Y = (Y_1, Y_2, \ldots, Y_N)$. Substituting in our choices of $l_i$ and $c_i$ in terms of $p_i$ and $s_i$ in Eq. (13), we obtain $\arg\min_{Y \in \{0, 1\}^N} \sum_{i=1}^N Y_i p_i$. Clearly, satisfying the storage constraint also satisfies the budget constraint in Knapsack by construction. Thus, the optimal solution to OMP as constructed gives the optimal solution to Knapsack. \qed