Title
Usable and Efficient Systems for Machine Learning

Author
Xin, Doris

Publication Date
2021

Peer reviewed|Thesis/dissertation

Usable and Efficient Systems for Machine Learning

by

Doris Xin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Aditya G. Parameswaran, Chair
Professor Joseph E. Gonzalez
Professor Joseph M. Hellerstein
Professor Niloufar Salehi

Spring 2021

Usable and Efficient Systems for Machine Learning

Abstract

Usable and Efficient Systems for Machine Learning

by

Doris Xin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Aditya G. Parameswaran, Chair

Machine learning has become a key driver for technological advancement in the last decade on the back of major progress in programming interfaces and scalable systems. Libraries such as Scikit-learn and Keras have made it easier to implement machine learning algorithms and applications, while innovations in distributed systems have enabled model training at an unprecedented scale. However, machine learning tooling is far from perfect today; practitioners still face many challenges when developing applications powered by machine learning.

This dissertation aims to improve the usability and resource efficiency of systems for developing and productionizing machine learning applications by investigating multiple directions identified through extensive empirical evidence gathering and analysis. First, we study the applied machine learning literature and execution traces of publicly available machine learning workflows to understand common practices and shed light on the highly iterative process of model development. Using our insights, we present a solution to accelerate the iterative model development process. Next, we analyze the provenance graph of thousands of production pipelines to uncover latent inefficiencies when running these pipelines. Using these insights, we propose a solution to significantly reduce wasted computation in such systems. Our solutions harness classic techniques from systems, databases, and programming languages to automate data management and optimize computation in machine learning application development. Finally, we synthesize findings from interviews with current users of automated machine learning tools to examine the role of automation in model development, as we look ahead to the future of machine learning developer tools.

To my mother.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without many great people. I am forever grateful that my advisor Aditya Parameswaran decided to take a chance on me when I approached him with a research idea outside his area of focus. Aditya has taught me a great deal about how to ask the right questions, how to be an effective communicator, and how to lead. Aditya has guided and inspired me to constantly strive to be better by setting a great example and by always holding me to higher standards. Aditya, thank you for showing me the way and for encouraging and supporting me to grow.

I would also like to thank my thesis committee members Niloufar Salehi, Joe Hellerstein, and Joey Gonzalez for guiding and inspiring me to embrace interdisciplinary research. Niloufar, thank you for introducing me to the wonderful world of HCI. Joe, your deep expertise in databases has propelled me to think more critically and be more rigorous. Joey, your expansive knowledge of machine learning and systems has been an inspiration for interdisciplinary innovations.

I started my graduate studies with Professor Jiawei Han. Professor Han is a pioneer, a visionary, and a great mentor. Working with him was a great privilege and honor, and I am grateful to have his blessing to pursue my passion.

Thank you to all my colleagues and collaborators over the years. I have learned a great deal from each and every one of you, and I am grateful for the opportunity to work with you all. Thank you to all my friends for always being there for me through thick and thin.

Most of all, I am eternally grateful for my mother's unconditional love and unwavering support.

Chapter 1

Introduction

Machine learning (ML) has become a staple in data-driven applications and continues to be a key driver of innovations transforming our everyday lives. Its rise to prominence in the technology landscape has been largely driven by major breakthroughs in developer experience and systems engineering. ML developers are now empowered with simple APIs to implement ML algorithms and applications through popular libraries such as Scikit-learn [174], Tensorflow [1], and PyTorch [148]. Software and hardware innovations such as parameter servers [112] and tensor processing units [86] have made it possible to train large models on massive datasets, which have been crucial for achieving high model accuracies. These advancements in developer experience and systems have enabled ML to become a mature technology that effectively serves a wide range of applications.

With the maturation of ML applications come new challenges around building and deploying ML models. Many have written about these challenges based on anecdotes or practical experience [222, 175, 11, 195, 102]. In this dissertation, we take a more rigorous approach to identifying challenges in developing and operationalizing ML models by gathering and analyzing empirical evidence from multiple sources, including applied ML literature, execution traces of user-generated ML workflows on an open ML platform, semi-structured interviews with ML practitioners, and provenance graphs of thousands of production ML pipelines at a large technology company. This series of need-finding studies shed light on existing practices around building and deploying ML models while exposing pain points and inefficiencies that deserve attention from tool developers. At the highest level, we find that

1. model development is inherently an iterative, human-in-the-loop process involving tedious, repetitive tasks;

2. ML applications in production require an exorbitant amount of computation to support, but a large portion of the computation consumed by existing ML systems have little impact on the application they support;

3. increasing interest in ML from a wide range of user personas pose profound questions on the role of automation in ML tooling.

We tackle many of the challenges we identify, using insights gathered from the need-finding stage to inform how we design the solutions. At the heart of our approaches to solve these challenges are reuse and redundancy minimization techniques to improve human efficiency and resource utilization efficiency, which can oftentimes benefit from co-optimization. Our solutions incorporate techniques from established fields, including databases, systems and programming languages, for data management and computation optimization in the ML setting.

In the rest of the chapter, we introduce common concepts used throughout the dissertation in Section 1.1 and Section 1.2, discuss the underpinning of our approach in Section 1.3, and provide an outline of the dissertation in Section 1.4.

1.1 The Machine Learning Workflow

A machine learning (ML) workflow, also commonly known as a machine learning pipeline[1], accomplishes a specific ML *task*, such as classification, entity recognition, and image captioning. It is a complex sequence of steps that, in turn, ingest raw input data, transforms it (often involving feature engineering) into a format amenable to model training, trains a model on the transformed data, and performs tasks using the trained model.

ML workflows commonly consist of three major components:

- **Data Preprocessing (DPR).** This stage contains all of the data manipulation operations, such as data cleaning and feature extraction, used to turn raw data into a format compatible with ML algorithms.
- **Modeling via Learning & Inference (L/I).** Once the data is transformed into a learnable representation, such as feature vectors, learning (L) takes place, using the transformed data to derive an ML model via optimization. Inference (I) refers to the process by which the learned model is used to make predictions on unseen data and is performed after learning.
- **Post Processing (PPR).** Post processing is the all-encompassing term for operations following learning and inference. Bruha et al. [30] classifies PPR operations into four categories: 1) rule-based knowledge filtering, 2) rule-based knowledge integration, 3) interpretation and explanation, 4) evaluation. While 1) and 2) involve transformations of the L/I output, 3) and 4) are about the analysis of the L/I output.

As we will demonstrate in Section 3.2.1 in Chapter 3, these three components are generic and sufficient for describing a wide variety of supervised, semi-supervised, and unsupervised settings. We will describe how we can represent an ML workflow as a directed acyclic graph of data *preprocessing* and ML *model* operators.

[1]We use the term *machine learning workflow* instead of machine learning pipeline in this dissertation to emphasize the complexity and non-linearity of the structure.

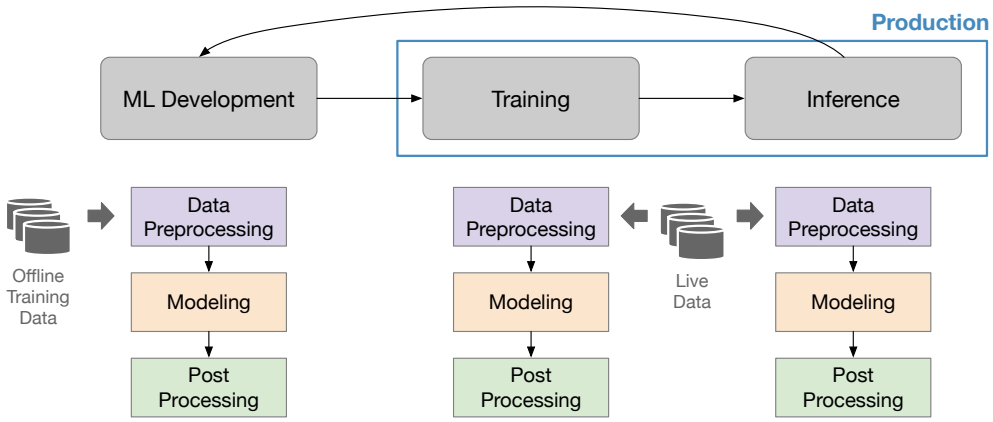


Figure 1.1: Machine Learning Lifecycle.

1.2 The Machine Learning Lifecycle

The ML lifecycle describes the process by which an ML workflow is developed and deployed to power applications. At a high level, the ML lifecycle comprises three stages, as shown in Figure 1.1. The first stage in the lifecycle is ML development, in which an ML workflow is developed offline through experimentation to accomplish a specific task. The developed workflow is then deployed into the production environment where it undergoes the next two stages in the lifecycle. The first production stage is training the model using the developed workflow on live production data, which is usually carried out in an environment with much more computation resources and on much larger scale data compared to the development stage. Finally, the trained model is deployed to downstream applications to perform inference on newly observed data.

ML Development. We start with our discussion on ML workflow development with a concrete example from our bioinformatics collaborators who form part of a genomics center at the University of Illinois [164].

Example 1 (Gene Function Prediction)**.** *The scientific goal of their ML application is to discover novel relationships between genes and diseases by mining scientific literature. To do so, they process published papers to extract entity—gene and disease—mentions, compute embeddings using an approach like word2vec [131], and finally cluster the embeddings to find related entities. They repeatedly iterate on this workflow to improve the quality of the relationships discovered as assessed by collaborating clinicians. For example, they may (i) expand or shrink the literature corpus, (ii) add in external sources such as gene databases to refine how entities are identified, and (iii) try different NLP libraries for tokenization and entity recognition. They may also (iv) change the algorithm used for computing word embedding vectors, e.g., from word2vec to LINE [184], or (v) tweak the number of clusters to control the granularity of the clustering. Every single change that they make necessitates*

waiting for the entire workflow to rerun from scratch—often multiple hours on a large server for each single change, even though the change may be quite small.

As the example illustrates, ML development is governed by an iterative process: starting with an initial workflow, developers iteratively modify their workflow, based on previous results, to improve performance. They may add or modify data sources, features, hyper-parameters, and training algorithms, among others. These iterations of trial-and-error are necessary due to data variability, algorithmic complexity, and overall unpredictability of ML.

In the context of ML development, an *iteration* involves creating a version of the workflow, either from scratch or by copying/modifying a previous version, and executing this version end to end to obtain some results for a specific task. Program termination marks the end of an iteration, and any results that are not written to disk during execution can only be obtained by modifying the workflow to explicitly save the results and rerunning the workflow. We explore the characteristics of ML development workloads by gathering and analyzing evidence from real-world development processes in Chapter 2 and propose a solution for accelerating ML development in Chapter 3.

Training. The final version of the workflow from the development stage is then run on a different dataset to train the model that will be deployed to downstream applications. This is a standard practice to avoid data leakage [92], i.e., the unintended side effect of introducing signals that are not part of the training data into model training. The model is usually trained on much more data than during development, wherein developers work with only a small sample for agile development.

As we will see in Chapter 4, in many production use cases, the model training process can be highly resource-intensive due to the scale of the full training data. Furthermore, the model is constantly updated on live data in order to combat model performance degradation due to data drift [64]. Given the enormous amount of computation resources required to support ML in production, resource utilization efficiency is paramount. In Chapter 5, we explore solutions to address a resource utilization efficiency issue uncovered in Chapter 4.

Inference. In the final stage of the ML lifecycle, the workflow produced from the training stage is deployed to support applications by performing inference on live data, often in real-time. A series of unique challenges arise in this setting due to the complexity and resource demands of the inference workload, many of which have been addressed by prior work [5, 40, 178]. This stage of the ML lifecycle is largely out of the scope of this dissertation.

1.3 Core Principles of Our Approach

The research conducted as part of this dissertation is guided by several core principles.

- First, we take a rigorous, evidence-based approach to discovering the needs in real-world use cases and use the results from the need-finding process to define the problems we tackle and design the solutions. By adhering to this principle, we can ensure that the outcome of the work in this dissertation has real-world impact.

- Second, we take a user-centric approach to solution design, in that the solutions should support existing practices rather than requiring drastic behavior changes from the users. We show via multiple instances in this dissertation that current deficiencies in the ML lifecycle can be addressed through better system design and implementation instead of requiring new practices on the part of the developers. We are able to accomplish this by conducting extensive need-finding prior to solution design and employing human-centered design practices.

- Finally, we recognize the fact that model training does not happen in a vacuum, and performance is highly dependent on other pieces in the ML workflow, such as data preprocessing [138]. Unlike the bulk of existing work on ML systems that focus solely on model training, we treat the end-to-end ML workflow as first-class citizens in our solutions. Considering the workflow end-to-end, as we show in Chapter 3 and Chapter 5, unlocks cross-component optimizations that lead to larger gains.

The common thread between our approaches to solving every system challenge in this dissertation is *reuse and redundancy minimization* techniques to improve human efficiency and resource utilization efficiency, which oftentimes benefit from co-optimization. By co-optimizing human efficiency and resource efficiency, we do not sacrifice one for the other. We incorporate established techniques from a number of disciplines in computer science, including databases, systems, and programming languages, for data management and computation optimization in the ML setting. We inject new life into these tried-and-true techniques by creatively adapting them to solve new challenges in machine learning tooling.

1.4 Dissertation Roadmap

The rest of the dissertation is organized into three parts, each containing a need-finding component and, optionally, solutions to address the needs discovered.

- The first part focuses on ML development and spans Chapters 2 (includes material from [210, 109]) and 3 (includes material from [209, 207]).

- The second part deals with ML in production and includes Chapter 4 and 5 (includes material from [211]).

- The solutions presented in the first two parts all involve the use of intelligent automation. Our final part, comprising Chapter 6 (includes material from [212]), examines the role of automation in ML tooling across the entire ML lifecycle, both as an introspective and as a vision for what is to come.

Chapter 2

Understanding Model Development

Machine learning model development has long been said to be a mysterious process—developers rely on iterative trial-and-error to obtain their own set of battle-tested guidelines to inform their modeling decisions. These guidelines can often seem arbitrary and paint a confusing picture of the underlying principles, if any, governing the process. Is there method to the madness? Can we arrive at a more concrete description of the iterative model development process?

To answer these questions and demystify the mysterious process that is ML model development, we conducted two studies. The first study involved a survey of the applied machine learning literature from five distinct application domains [210]. We use statistics collected from the papers to understand the model development process and report preliminary trends and insights from our investigation. This study served as a lens into how experts experiment with model building. In the second complementary study, we study a wider population of predominantly citizen data scientists [109]. We analyze over 475k user-generated workflows on OpenML, an open-source platform for tracking and sharing ML workflows, to identify popular practices as well as their efficacy.

Together, the results from the two studies present a detailed, statistical characterization of how developers iteratively modify ML workflows, which can serve as a benchmark for machine learning workflow development in practice, and can aid the development of future human-in-the-loop machine learning systems. This is a step forward from the currently popular approach of using anecdotal evidence to identify usage patterns and motivate design decisions, such as in Kumar et al. and Zhang et al. [101, 222]. Based on our findings, we finally describe desiderata for effective and versatile human-in-the-loop machine learning systems that can cater to users in diverse domains.

2.1 A Survey of the Applied Machine Learning Literature

We conduct a statistical study of iteration by surveying the applied ML literature across five application domains. The statistics collected in this study provide the first quantitative evidence of how developers iterate on ML workflows, beyond anecdotal ones. Moreover, the insights and trends discovered from our survey provide concrete guidelines on desired human-in-the-loop ML system properties, while the models and statistics provide a starting point for the development of benchmarks for standardized and automatic evaluation of human-in-the-loop ML systems.

Statistical studies of end-to-end ML workflow development pose several challenges. First, it is difficult to gather data that captures the entire process, and not just the final snapshot. One approach, for example, may involve examining code repositories over time to determine what has changed—one downside of this approach is that developers may not commit intermediate iterations, leading to less transparency for the overall process. Moreover, this approach will require understanding code, and mapping code fragments to classes of iterative modifications, both of which are extremely challenging to do. Second, we need to ensure that our study captures a diverse set of application domains. Surveys [94, 185, 118, 137] often end up focusing on industry-relevant application areas (e-commerce, recommendations), and data-types (language, vision). Since our eventual goal is to develop a benchmark for general-purpose human-in-the-loop ML systems, this limited view may hinder our ability to adequately support all application domains. Third, once the data is collected, we need to devise methods to analyze the data and collect statistics related to iteration. Finally, we need to turn the raw statistics into models that capture iteration and relate trends and insights discovered from these models to ML system design.

Our study includes an analysis of 105 applied machine learning papers sampled from multiple conferences in 2016 and across five application domains, including social sciences, natural sciences, web application, computer vision, and natural language processing. We collect statistics from each paper that capture iterative development and use these statistics to infer common practices in each application domain surveyed. We describe the statistics collected, how they are used to estimate iteration counts, and discuss the limitations of our approach in the next section. To ensure the quality of our statistics, we take consensus over results collected by multiple surveyors, and open-source the final aggregated data for further studies by interested readers, as well as development of formal benchmarks.

2.1.1 Related Work

To the best of our knowledge, our survey is the first effort in conducting a statistical study of machine learning model development from empirical evidence. However, the pursuit of understanding iterative ML development is not singularly ours. Several surveys have been conducted in recent years to profile industry and academic ML users [94, 185, 118, 137].

These surveys differ from ours in that they were self-reported responses from a select set of industry and academic users. Findings from self-reporting surveys are known to suffer from response bias [139]. Many articles discuss general trends and design patterns in ML workflows [51, 85, 119], while a number of articles focus on providing guidance and taxonomies for novice users to perform iteration better [223, 159, 170]. Other work such as [2] and [114] study general trends and needs in data science using NLP techniques to study a large corpus en masse. Vartak et al. [195] describe a system-building vision for iterative human-in-the-loop ML. Kery et al. [93] specifically study the versioning aspect of iterative development, whereas Koesten et al. [97] analyze in-depth surveys to understand the typical workflow for data scientists.

2.1.2 Data & Methodology

In this section we describe the dataset and the methods used to collect the statistics that enable analyses of iteration in publications.

2.1.2.1 Corpus

We surveyed 105 papers published in 2016 on applied data science. To ensure relevance, we selected four venues that specifically publish applied machine learning studies: KDD Applied Data Science Track, Association for Computational Linguistics (ACL), Computer Vision and Pattern Recognition (CVPR), and Nature Biotechnology (NB). We randomly sample 20 papers from ACL, CVPR, and NB each, and 45 papers from KDD. These papers span applications in social sciences (SocS), web applications (WWW), natural sciences (NS), natural language processing (NLP), and computer vision (CV). Paper topics were determined using the ACM Computing Classification System (CCS)[1]. Keywords in each paper are matched with entries in the CCS tree, and each paper is assigned as its domain the most appropriate high level entry containing its keywords. Figure 2.1 illustrates the domain composition of the conferences surveyed. While ACL, CVPR, and Nature specialize in a single domain, KDD embraces many domains, with a focus on web applications and social science.

Limitations. Our approach is limited in its ability to accurately model iterations due to several characteristics of the corpus:

1) While the corpus spans multiple domains, the number of papers in each domain is small, which can lead to spurious trends.
2) Papers provide an incomplete picture of the overall iterative process. Machine learning papers are results-driven and focus more on modeling than data preprocessing by convention. Due to space constraints, authors often omit a large number of iterative steps and report only on the small subset that led to the final results.

[1] https://www.acm.org/publications/class-2012

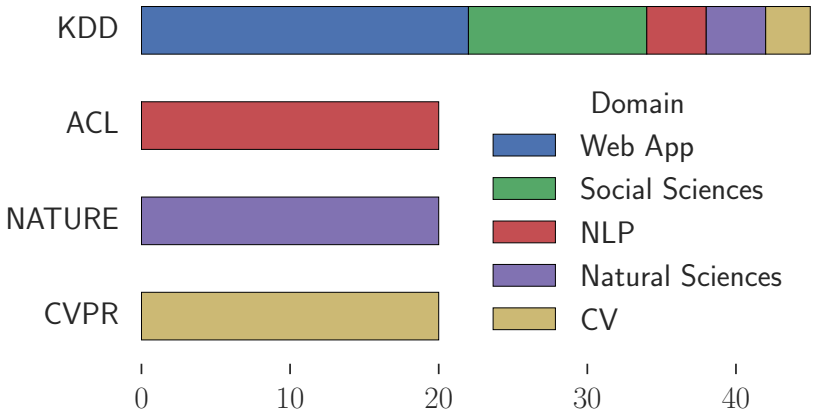


Figure 2.1: Paper count per domain by conference.

3) Papers often present results side by side instead of the order they were obtained, making it difficult to determine the exact transitions between the variants studied in the iterative process.

We attempt to overcome some of these limitations by

- Having multiple surveyors and aggregating the results to reduce the change of spurious results, to be elaborated in Section 2.1.2.2;
- Devising estimators that do no rely on information about the order of operations, to be elaborated in Section 2.1.2.3.

2.1.2.2 Statistics Collection

Our goal in this survey is to collect statistics on how users iterate on ML workflows. However, iterations are often not explicitly reported in publications. To overcome this challenge, we design a set of statistics that allow us to infer the iterative process leading to the results reported in each paper. We introduce the statistics for each individual component of the ML workflow below.

Data Preprocessing (DPR). DPR encompasses all operations involved in transforming raw data into learnable representations, such as feature engineering, data cleaning, and feature value normalization. We record $\mathcal{D}$, the set of distinct DPR operation types found in each paper and collect $n_{\mathcal{D}} = |\mathcal{D}|$. Mentions of DPR operations are usually found in the data and methods sections in the paper.

Learning/Inference (L/I). Workflow modifications concerning L/I fall into one of three categories: 1) hyperparameter tuning for a model (e.g., increasing learning rate, changing the architecture of a neural net) and 2) switching between model classes (e.g., from decision tree to SVM). For each paper, we record $\mathcal{M}$, the set of all model classes and $\mathcal{P}$, the set of distinct hyperparameters tuned across all model classes, and collect $n_{\mathcal{M}} = |\mathcal{M}|$ and $n_{\mathcal{P}} = |\mathcal{P}|$.

Evidence for these statistics is usually found in the algorithms section, as well as result tables and figures.

Post Processing (PPR). Evaluation and interpretation/explanation are the most commonly reported post-processing operations in papers, often presented in tables or figures. For each paper, we record $\mathcal{E}$, the set of evaluation metrics used, and collect $n_{\mathcal{E}} = |\mathcal{E}|$. In addition, we collect n_{table} and n_{figure}, the number of tables and figures containing results and case studies, respectively.

We refer to $\mathcal{D}, \mathcal{M}, \mathcal{P}, \mathcal{E}$ collectively as *entity sets* in the rest of the paper [2].

To ensure the quality of the statistics collected, we had three graduate students in data mining, henceforth referred to as *surveyors*, perform the survey independently on the same corpus. We reference the results collected by each surveyor with a subscript, e.g., $\mathcal{M}_1$ is the set of model classes recorded by surveyor 1. To increase the likelihood of consensus, we first had the surveyors discuss and agree on a seed set for each entity set, e.g., $\mathcal{E} = \{$Accuracy, RMSE, NDCG$\}$. Surveyors were then asked to remove from and add to this set as they see fit for each paper. Let n'_x be the aggregated value of the statistic n_x. We aggregate the three sets of results as follows:

- For an entity set S (e.g., $\mathcal{M}$, the set of model classes), let $S_a = S_1 \cup S_2 \cup S_3$. We filter S_a to obtain $S' \subseteq S_a$ such that $s \in S'$ is identified by at least two surveyors. That is, a paper is considered to contain an operation only if it is identified to be in the paper by at least two surveyors independently. We define n'_S for the corresponding statistic as $|S'|$.
- For n_{table} and n_{figure}, we define $n'_{table/figure}$ to be the average of the values obtained by the three surveyors.

2.1.2.3 Estimating Iterations using Statistics

The information collected above indicate versions of the workflow studied but not the iterative modifications themselves. To infer the number of iterations using the statistics collected above, we make the following assumptions:

- Each iteration involves a single change. While it is possible for multiple changes to be tested in a single iteration, it is unlikely the case since the interactions can obfuscate the contribution of individual changes.
- Each element in an entity set is tested exactly once. For the authors to report on a variant, there must have been at least one version of the workflow containing that variant. Although it is likely for a variant to be revisited in multiple iterations in the actual research process, papers, by convention, provide little information on this aspect. Due to this lack of evidence, we take the conservative approach by taking the minimum value.

[2] The complete entity sets and statistics can be found at https://github.com/gestalt-ml/AppliedMLSurvey/blob/master/data/combinedCounts.tsv

Let t_{DPR}, t_{LI}, t_{PPR} be the number of iterations containing changes to the DPR, L/I, and PPR components of the workflow, respectively. Using the two assumptions above, we estimate t_{DPR}, t_{LI}, and t_{PPR} as follows:

- $\hat{t}_{DPR} = n'_{\mathcal{D}}$
- $\hat{t}_{LI} = (n'_{\mathcal{M}} - 1) + (n'_{\mathcal{P}} - 1)$
- $\hat{t}_{PPR} = \min\left(n'_{\mathcal{E}}, n'_{table} + n'_{figure}\right)$

For $\hat{t}_{DPR}$, we assume that the authors start with the raw data and incrementally add more data preprocessing operations in each iteration. We subtract one from $n'_{\mathcal{M}}$ and $n'_{\mathcal{P}}$ in $\hat{t}_{LI}$ to account for the fact that the initial version of the workflow must contain a model, a set of hyperparameters, and an optimization algorithm. The estimator $\hat{t}_{PPR}$ assumes that in a PPR iteration, the authors can either gather all information on a single metric or generate an entire figure/table.

2.1.3 Results and Insights

In this section we share interesting trends about ML workflow development discovered from our survey.

2.1.3.1 Iteration Count

Figure 2.2 shows the histograms for the three iteration estimators $\hat{t}_{DPR}, \hat{t}_{LI}, \hat{t}_{PPR}$ across the entire corpus (top row) and by domain (rows 2-6). A bin in every histogram represents an integral value for the estimators, and bin heights equal the fraction of papers with the bin value as their estimates. The mean values for the estimators by domains are shown in the stacked bar chart in Figure 2.3, where the total bar length is equal to the average number of iterations in each domain. From these two figures, we see that 1) most papers use ≥ 1 evaluation methods, evident from the fact that histograms in the third column in Figure 2.2 are skewed towards $\hat{t}_{PPR} \geq 2$; 2) PPR is the most common iteration type across all domains, evident from the length of the $\mathrm{E}[\hat{t}_{PPR}]$ bars in Figure 2.3; and 3) on average, more DPR iterations are reported than L/I iterations in every domain except computer vision, as illustrated by the relative lengths of the $\mathrm{E}[\hat{t}_{DPR}]$ and $\mathrm{E}[\hat{t}_{LI}]$ bars in Figure 2.3.

When grouped by domains, we see that the distributions for certain domains deviate a great deal from the overall trends in Figure 2.2. Domains dominated by deep neural nets (DNNs), which are designed to replace manual feature engineering for higher order features, tend to skew towards fewer DPR and more L/I iterations, such as NLP and CV. Additionally, there are only a few highly processed datasets studied in all NLP and CV papers, further reducing the need for data pre-processing in these domains. On the other hand, social and natural sciences exhibit the opposite trend in the histograms in Figure 2.2, biasing towards more DPR iterations. This is largely due to the fact that both domains rely heavily on domain knowledge to guide ML and strongly prefer explainable models. In addition, a large amount of data is required to enable training of DNNs. The scale of data is often much

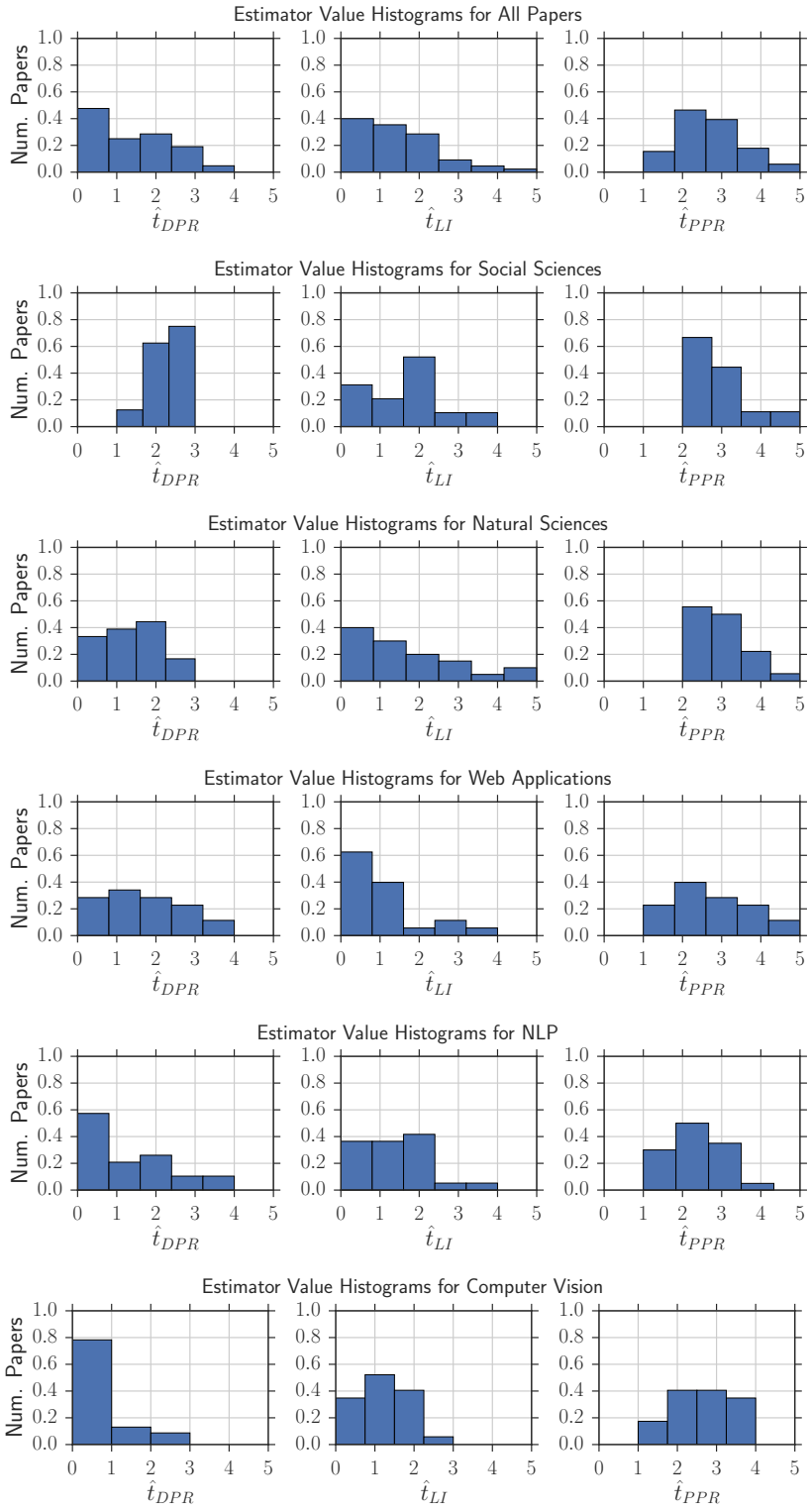


Figure 2.2: Distribution of number of iterations by workflow component.

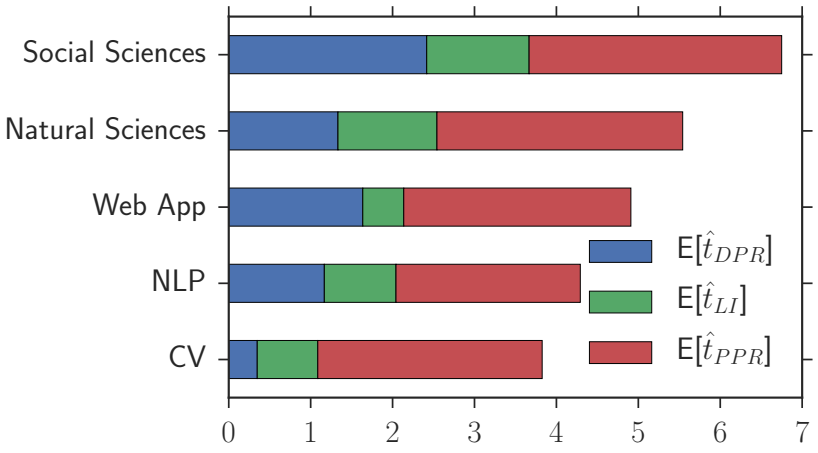


Figure 2.3: Mean iteration count by domains.

smaller for SocS and NS than NLP and CV, thus preventing effective application of DNNs and requiring more manual features.

2.1.3.2 Data Pre-processing by Domain

Table 2.1 shows the most popular DPR operations in each application domain, ordered top to bottom by popularity, with abbreviations expanded in the caption. While the table reaffirms common knowledge such as feature normalization is important, Table 2.1 also shows two striking results: 1) joining multiple data sources is common in four of the five domains surveyed; 2) $\frac{1}{3}$ of the papers contain fine-grained features defined using domain knowledge across all domains. Result 1) suggest that unlike classroom and data competition settings in which the input data resides conveniently in a single file, data in real-world ML applications is aggregated from multiple sources (e.g., user database and event logs). Result 2) contradicts the common belief that ML applications have collectively progressed beyond handcrafted features thanks to the advent of deep learning (DL). In addition to the incompatibilities with DL in some domains mentioned in Section 2.1.3.1, the efficacy of features designed using domain knowledge versus using DL to search for the same features without domain knowledge is possibly another contributing factor.

2.1.3.3 Learning/Inference by Domain

Table 2.2 lists the most popular model classes for each application domain, with abbreviations expanded in the caption. We have already discussed the disparity between the popularity of DL in CV/NLP and other domains in Section 2.1.3.1. Most traditional approaches such as GLM, SVM, and Random Forest are still in favor with most domains, since the large additional computation cost for DL often fails to justify the incremental model performance gain. Matrix factorization, which is highly amenable to parallelization, is popular in web

SocS	NS	WWW	NLP	CV
Join (31.0%)	Feature def. (40.6%)	Feature def. (36.1%)	Feature def. (32.1%)	Feature def. (37.5%)
Feature def. (27.6%)	Univar. FS (18.8%)	Join (22.2%)	BOW (17.9%)	BOW (25.0%)
Normalize (17.2%)	Normalize (12.5%)	Normalize (13.9%)	Join (14.3%)	Interaction (25.0%)
Impute (6.9%)	PCA (9.4%)	Discretize (8.3%)	Normalize (10.7%)	Join (12.5%)

Table 2.1: Common DPR operations ordered top to bottom by popularity. Join = joining multiple data sources; Feat. def. = custom logic for fine-grained feature extraction; Univer. FS = univariate feature selection, using criteria such as support and correlation per feature; BOW = bag of words; PCA = principal component analysis, a common dimensionality reduction technique.

SocS	NS	WWW	NLP	CV
GLM (36.0%)	SVM (32.7%)	GLM (37.0%)	RNN (32.4%)	CNN (38.2%)
SVM (28.0%)	GLM (15.4%)	RF (11.1%)	GLM (14.7%)	SVM (17.6%)
RF (20.0%)	RF (13.5%)	SVM (11.1%)	SVM (11.8%)	RNN (17.6%)
Decision Tree (12.0%)	DNN (13.5%)	Matrix Factorization (11.1%)	CNN (8.8%)	RF (5.9%)

Table 2.2: Common model classes ordered top to bottom by popularity per domain. GLM = generalized linear models (e.g., logistic regression); RF = random forest; SVM = support vector machine; R/CNN = recursive/convolutional neural networks.

SocS	NS	WWW	NLP	CV
Regularize (40.0%)	CV (31.8%)	Regularize (41.2%)	LR (39.4%)	LR (46.2%)
CV (30.0%)	LR (22.7%)	LR (23.5%)	Batch size (24.2%)	Batch size (30.8%)
LR (10.0%)	DNN arch. (18.2%)	Batch size (11.8%)	DNN arch. (18.2%)	DNN arch. (11.5%)
Batch size (10.0%)	Kernel (9.1%)	CV (11.8%)	Kernel (6.1%)	Regularize (11.5%)

Table 2.3: Most popular model tuning operations by domain. CV = cross validation; LR = learning rate; DNN arch. = DNN architecture modification; Kernel specifically applies to SVM.

SocS	NS	WWW	NLP	CV
P/R (25.7%)	Acc. (28.6%)	Acc. (20.8%)	P/R (29.2%)	Vis. (33.3%)
Acc. (20.0%)	P/R (18.6%)	P/R (20.8%)	Acc. (27.1%)	Acc. (29.8%)
Feat. Contrib. (17.1%)	Vis. (15.7%)	Case (13.2%)	Case (14.6%)	P/R (17.5%)
Vis. (14.3%)	Correlation (11.4%)	DCG (9.4%)	Human Eval. (8.3%)	Case (12.3%)

Table 2.4: Most popular evaluation methods by domain. P/R = precision/recall; Acc. = accuracy; Vis. = visualization; Feat. Contrib. = feature contribution to model performance; NCG = discounted cumulative gain, popular in ranking tasks; Case = case studies of individual results.

applications for supporting recommendation engines. Interestingly, SVM is the most popular method in natural sciences by a large margin (100% more popular than the second most popular option), possibly due to its ability to support higher order functions through kernels. NS applications experimenting with DL are mostly computer vision related.

Table 2.3 shows the most popular model tuning operations by domains. The top two operations, learning rate and batch size, are both concerned with the training convergence rate, suggesting that training time is an important factor in all domains. Cross validation and regularization are both mechanisms to control model complexity and overfitting to observed data. Lower complexity models usually result in faster inference time and better ability to generalize to more unseen data.

2.1.3.4 Post Processing by Domain

Of the evaluation methods listed in Table 2.4, P/R, accuracy, correlation, and DCG are summary evaluations of model performance while case study, feature contribution, human evaluation, and visualization are fine-grained methods towards insights to improve upon the current model. While the former group can be used automatically such as in grid search, the latter group is aimed purely for human understanding.

2.2 Analysis of Machine Learning Workflow Execution Traces

The study in this section complements the one in the preceding section in two important ways: 1) while the study in the last section shed light on how experts develop ML models, the study in this section offers a complementary view of the process by analyzing ML workflow development behavior of users with a wider range of expertise; 2) the previous study relies on results that have been curated and selectively reported by the developer, whereas the dataset in this study provides a comprehensive view of every iteration carried out by the developer in the model building process.

The data used in this study is gathered from OpenML, which is an open-source, hosted platform for users to upload datasets and run ML workflows on these datasets by calling an API. A relatively diverse mix of user skill levels is present on OpenML, from students just getting started with ML to experienced data scientists and ML researchers. OpenML publishes a database of the user-uploaded datasets as well as the workflow specifications submitted by users and their corresponding executions. By performing targeted analyses on the most common ML models and preprocessing operators as well as their associated performance, we shed light on common ML workflow design patterns and their general effectiveness. We study trends in iterative ML workflow changes across 295 users, 475,297 runs, and 793 tasks on the OpenML platform.

The main contributions in this study can be summarized as follows:

- We characterize the frequency and performance of popular ML models and preprocessing operators used by OpenML users. We highlight the impact of the operator combinations and discuss their implications on general user awareness (or lack thereof) of particular ML concepts (Section 2.2.3).
- We analyze sequences of changes to workflows to extract different styles of ML workflow iteration and shed light on the most common types of changes for each iteration style, the amount of exploration typically performed, and performance gain users are generally able to achieve (Section 2.2.4).
- We conduct case studies on exemplary instances to understand effective iteration practices. (Section 2.2.5).

2.2.1 Related Work

Psallidas et al.[158] analyzed publicly-available computational notebooks and enterprise data science code and pipelines to illustrate growing trends and usage behavior of data science tools. Other studies have employed qualitative, semi-structured interviews to study how different groups of users engage with ML development, including how software engineers [11] and non-experts [213] develop ML-based applications, and how ML practitioners iterate on their data in ML development [78]. In comparison, we analyze practices and behavior for users iterating on ML workflows, based on data collected from OpenML [192]—an online platform for organizing and sharing ML experiments to encourage open science and collaboration. Data from OpenML has been used extensively for meta-learning [191] to recommend optimal workflow configurations, such as preprocessing [25, 157], hyperparameters [190], and models [24, 180] for a given dataset and task. Benchmark datasets from OpenML, as well as other similar dataset repositories [140, 43], have also been used for evaluating AutoML algorithms [66, 34, 63]. However, these papers focus solely on using the datasets, models and results from each workflow and do not study the iterative behavior of users.

This study differs from existing work in that we analyze the trace of *user-generated* ML workflow iterations leading to insights such as which stages (preprocessing, model selection, hyperparameter tuning) users focus most of their iterations on, their impact on the effectiveness of the workflow, and which specific combinations of model and preprocessing operators are the most widely used. Moreover, our study offers a complementary perspective to existing interview studies by providing empirical (based on code), population-level statistics and insights on how people iterate on machine learning workflows. These insights are valuable for helping AutoML and mixed-initiative-ML system builders understand the trends in the ML development practices of target users.

2.2.2 Data & Methodology

Recall from Section 1.1, an ML *workflow* is a directed acyclic graph of data *preprocessing*, ML *model* with their associated hyperparameters, and post processing operators. A workflow

is created to accomplish a specific *task*, which consists of a dataset along with a machine learning objective, such as clustering or supervised classification. A *run* is the execution of a workflow applied to a particular task (multiple executions of the same workflow lead to multiple runs), and a *sequence* consists of the time-ordered set of runs from a single user for a particular task. In the OpenML dataset, workflows contain only data preprocessing and model operators, and each run is associated with performance measures, such as classification accuracy or Area Under the ROC Curve (AUC).

2.2.2.1 The Dataset

Our dataset is derived from a snapshot of the OpenML database from December 4, 2019. We focus on a specific subset of the runs on OpenML that a) were uploaded by users who are not OpenML developers[3] or bots[4], to focus on realistic human user behavior, b) use the Scikit-Learn package [150], to increase the fidelity of our data extraction by focusing on a single popular ML package used in 79% of the non-developer/non-bot runs, and c) have an associated AUC score associated with them. This subset contains 475,297 runs (4.8% of the total number of runs)

2.2.2.2 Workflow Effectiveness Metrics

Due to the variability in the range of evaluation metric values across tasks, raw AUC values of runs are not directly comparable across tasks. Instead, we account for the difficulty of each task by measuring *relative AUC*. For a run r, let a_r be its raw AUC on the corresponding task t. The relative AUC of a run r, is defined as $p_r = a_r - \mu_t$, where μ_t is the average AUC of all of the runs for task t. We measure the performance of a sequence S by the *relative maximum sequence AUC*, $p_S = \max_{r \in S} a_r - \frac{1}{|\mathcal{T}_t|} \sum_{Q \in \mathcal{T}_t} \max_{q \in Q} a_q$, where $\mathcal{T}_t$ is the set of all sequences for task t. In other words, p_S is the difference between the maximum AUC in the sequence S and the mean of the maximum AUCs of all sequences for task t. We use maximum AUC per sequence to capture the fact that the best-performing workflow out of all attempts is the final one that the user adopts.

2.2.3 Run-Level Insights

To better understand how users develop ML workflows, we first sought to understand: *What are the most prevalent ML operators in practice? In what combinations are these operators used together and when do they lead to better performance?*

Figure 2.4 shows the most common model and preprocessing combinations used by more than four users on OpenML. Ensemble models that combine one or more base estimators were

[3]We filtered out runs uploaded by the core team and key contributors listed on `https://www.openml.org/contact`.

[4]We also filtered out runs uploaded by users whose names contained "bot." Removing developers and bots left us with 6.1% of the total number of runs on OpenML.

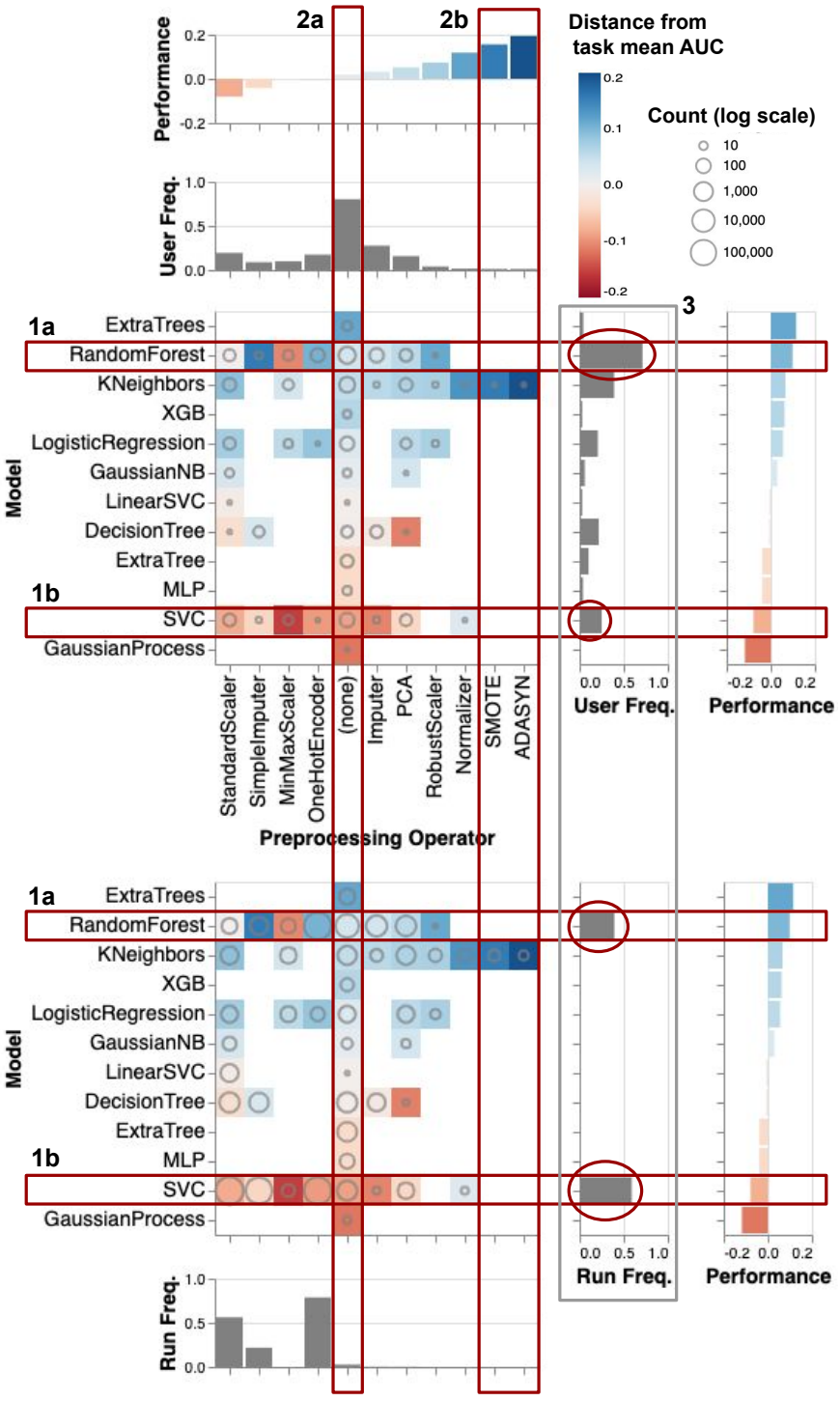


Figure 2.4: Frequency and performance of the most common (Scikit-Learn model, preprocessing) combinations in OpenML. The top displays frequency in terms of the user count, while the bottom shows run count.

excluded from the plot, to avoid duplication with other base estimators. DummyClassifier was also excluded. The color indicates the performance of the combination, measured by p_r, averaged over all runs that include the combination, while the size of each circle is scaled by the number of users (Figure 2.4 top) and runs (Figure 2.4 bottom) that contain the specific combinations. Both the rows and columns are sorted based on their performance across all of the specified operators. In other words, the best-performing combinations are located at the top-right corner of the chart, whereas the worst-performing combinations are on the bottom-left.

Certain models (AdaBoostClassifier, BaggingClassifier, DummyClassifier, GradientBoostingClassifier, GridSearchCV, RandomizedSearchCV, and VotingClassifier) were intentionally excluded from the plot. DummyClassifier was excluded because it is known to be used for comparisons rather than for solving real problems [150]. The others were not shown because they are wrappers that can take in one or more base estimators, and the most frequently used of these base estimators are already shown in the plot.

Histograms showing the marginal distribution of frequency and performance for each operator are displayed on the sides. For instance, the uppermost user frequency histogram shows the average user count for each preprocessing operator, averaged across the models that were used in combination with it, normalized by the total number of users.

We highlight various insights from Figure 2.4, with the enumerated points corresponding to the enumerated boxed regions in the figure.

(1) Performance of Specific Combinations: Some combinations consistently yield high relative AUC when compared to other combinations used for the same ML task. For instance, there is a clear difference in performance between Random Forest (1a), the model used by the most OpenML users, and SVC (C-Support Vector Classification) (1b), the model used in the most runs. On average, the relative AUC of runs that include Random Forest (RF) is $p_r = 9.89\%$. RF works especially well with SimpleImputer [5] as an added preprocessing step, achieving $p_r = 15.57\%$ on average. On the other hand, users tend to perform worse on average ($p_r = -7.84\%$) when using SVC models for OpenML tasks.

(2) Effect of Preprocessing: Around 81% of users have run ML models on a dataset without performing any data preprocessing beforehand (2a). This could be attributed to the fact that many of the datasets on OpenML are already in a relatively clean, preprocessed state. However, it is evident from Figure 2.4 that users can often still achieve higher performance by including some form of dimensionality reduction, feature scaling and transformations, or data sampling, indicating how *preprocessing is often an overlooked but important aspect in ML development*. For instance, ADASYN and SMOTE [111], which are preprocessing strategies for over-sampling data, are relatively infrequent among OpenML users (2b). However, when combined with K-Nearest Neighbors (KNN) models, users are able to consistently achieve higher AUC scores (by around $p_r = 19.4\%$ for ADASYN and $p_r = 15.6\%$ for SMOTE). This suggests that when working with imbalanced datasets, certain techniques such as oversampling are extremely valuable in achieving high classification performance, and that there

[5] https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html

Model	Run Freq	User Freq	Avg p_r
DecisionTree	**65.64**%	6.06%	0.57%
KNeighbors	19.02%	**60.61**%	0.48%
RandomForest	9.82%	18.18%	3.1%

Table 2.5: Frequency and average distance from task mean AUC of the most commonly used models for large datasets.

needs to be increased awareness of such data preprocessing methods.
(3) Frequency of Specific Combinations: The frequency distributions for both models and preprocessing operators vary depending on whether we look at the number of runs or the number of users (shown in box 3). For example, the most popular model measured by the number of unique users is RF (used by 68.2% of users in our dataset) followed by KNN (37.4% of users), with SVC in third place (24.2% of users). However, when determining frequency from the perspective of the number of runs that included the model, SVC makes up $\approx 50\%$ of the runs in our dataset, while RF accounts for $< 40\%$ of the runs. This suggests that there are variations in the iteration behavior across different users: some focus on tuning the same model for many iterations, while others experiment with several different models. We explore these trends and provide insights on their effectiveness in the Section 2.2.4.

2.2.3.1 Application of Run-Level Insights

Analyses such as those highlighted from (1)-(3) not only shed light on the usefulness of particular ML operators, but they can also be used to validate whether users' existing practices aligns with *conventional wisdom* in the form of guidelines from the ML community or whether there is a gap in adopting these guidelines. As an example of this application, we examine a particular case study of how users select which models to use for handling large datasets.

First of all, we observe that for large datasets, users do indeed focus on specific models in a different distribution than the overall trends shown in Figure 2.4. Table 2.5 shows the model frequencies for only the runs on large datasets (with greater than 110,313 instances—the mean across all the datasets that Scikit-Learn users constructed workflows for).

According to conventional guidelines [205], Decision Trees and ensemble methods like RF are well-suited for medium to large datasets, while KNN works well for small to medium datasets. Although Decision Tree makes up 65.64% of the runs, it is used by only 6% of the users. Instead, 60% of the users opted for KNN for the largest datasets on OpenML. However, KNN resulted in the lowest performance, (average $p_r = 0.48\%$), compared to average $p_r = 3.1\%$ for RF and 0.57% for Decision Tree. While the performance validates the efficacy of the guidelines, the usage statistics reveal that most users fail to follow these guidelines.

To better understand how users construct their workflows, next, we delve into how users

iterate on their workflows and the impact of different aspects of these iterative changes on the performance of the workflows.

2.2.4 Sequence-Level Insights

Users have a wide range of ML workflow development styles—some use a more manual approach where they run a model, look at the results, make a change or two to address the issue, and then repeat the process. Others may choose a more automated technique, e.g., looping through a set of pre-determined values for certain hyperparameters of a model. In the *manual* case, the human remains *in the loop*, while in the *automated* case, the user has already set a search space a-priori and the changes to the workflow at each iteration are independent of the previous iteration's result. Others use a *mixed* sometimes-manual, sometimes-automated strategy.

To classify a sequence as manual, automated, and mixed, we introduce the following metrics.

- Interval (Δt): difference between start times of consecutive runs
- Interval difference ($\Delta^2 t$): difference between consecutive Δts
- Sequence length ($|S|$): the number of runs in sequence S

Based on these metrics, we categorize each sequence S as follows:

- **Manual**: ($|S| \leq 2$, OR $\Delta t > 10$ minutes for $\geq 50\%$ of the runs in the sequence, OR $\Delta^2 t > 3$ minutes for $\geq 75\%$ of the runs in the sequence), AND $|S| < 300$
- **Automated** (Auto): $|S| > 2$, AND $\Delta t > 10$ minutes for $< 50\%$ of the runs in the sequence, AND $\Delta^2 t > 3$ minutes for $\leq 25\%$ of the runs in the sequence
- **Mixed**: the remaining sequences not in the two categories above

The thresholds were empirically determined through a process of random sampling and spot-checking two sequences from the manual category that had over 30 iterations and two from the automated category that had under 30 iterations (since these would be the more ambiguous cases than shorter manual sequences and longer auto sequences). After the final thresholds were set, five sequences from each of the categories were randomly sampled and spot-checked to validate the sequence labels.

The motivation behind looking at both Δt and $\Delta^2 t$ is that we would expect the actual time difference between run submissions to be at least a couple of minutes if the user was making adjustments manually, and we would expect the change in these Δt's to also be non-zero due to the variability in making changes (while constant time differences are often indicative of a loop). Sequence length is also highly indicative of whether or not most of the runs were manual, since it would be unlikely to find hundreds or thousands of manual runs, and indeed the highest number of runs in a manual sequence (after setting the 300 iteration threshold) is 69. This categorization results in 2181 manual, 208 automated, and 168 mixed sequences. The total number of runs in each group (sum of sequence lengths in the category) are: 7,059 manual, 454,270 auto, and 13,968 mixed.

Overall, we observe that users who adopt a more hands-on approach achieve good performance much faster than users who automate the search. Across the three categories, users

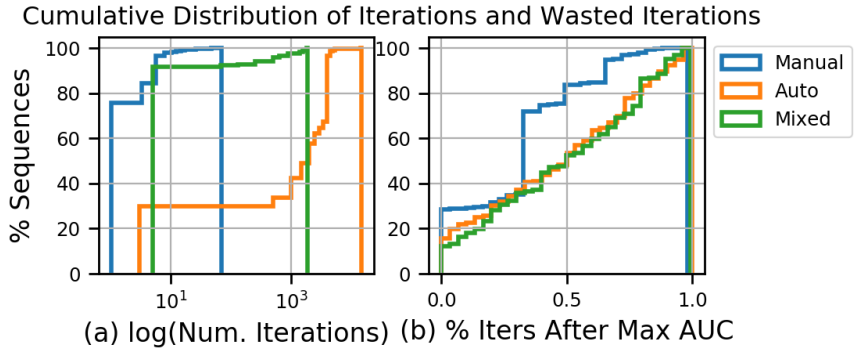


Figure 2.5: (a) Cumulative distribution of sequence length, (b) % iterations after reaching maximum AUC.

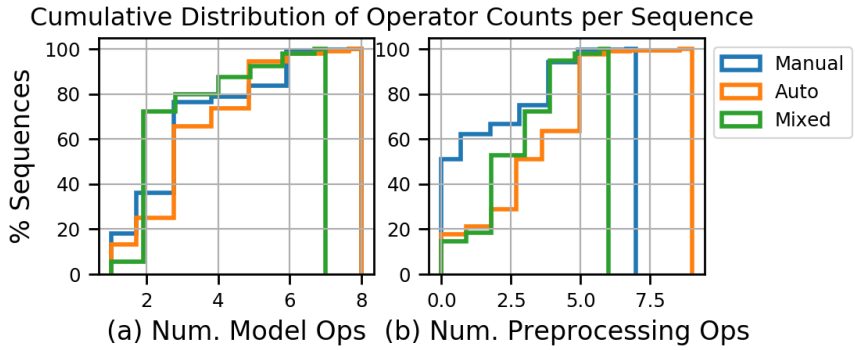


Figure 2.6: Cumulative distribution of (a) model and (b) preprocessing operators used across all iterations in a sequence.

exert varying amounts of effort (number of iterations and number of model and preprocessing combinations attempted) and focus on different areas of their workflow (choosing the best model, optimizing a hyperparameter, or determining which data preprocessing operation to add). Our in-depth analyses of the three categories of sequences reveal three major insights on the effectiveness of users iterating with manual and automated iterations, described next.

2.2.4.1 On Efficiency

Users iterating with manual sequences are more efficient, achieving the same performance gain in a small fraction of the number of iterations as automated sequences, while wasting fewer iterations searching after reaching their highest-performing workflow. However, users iterating with automated sequences reach a higher p_S than manual for the same task.

Figure 2.5(a) shows that the distribution of the number of iterations per sequence is drastically different for each sequence category. Manual sequences are short ($\mu = 3.24, \sigma = 3.88$) and automated sequences are the longest ($\mu = 2183.99, \sigma = 1953.74$), while mixed

Change Type	Manual	Mixed	Auto
Model Operator	**45.04**%	3.95%	0.18%
Model Hyperparameter	28.31%	**75.23**%	**92.69**%
Preprocessing Operator	0.53%	0.55%	0.01%
Preprocessing Hyperparameter	0.18%	0.17%	0.04%
Model & Preprocessing	21.96%	6.60%	5.45%
No Change	3.98%	13.49%	1.63%

Table 2.6: Change types for manual, mixed, and automated.

lies somewhere in-between ($\mu = 83.14, \sigma = 283.98$). Even though manual sequences are on average $< 0.2\%$ the length of automated sequences, within the same span of time, the maximum increase in AUC is equal for manual and automated (manual: $\mu = 6.63\%, \sigma = 8.65\%$; auto: $\mu = 7.06\%, \sigma = 12.66\%$)[6]. Moreover, Figure 2.5(b) shows that the manual group has a lower percentage of wasted iterations, i.e., the iterations after p_S has been achieved, than the other two groups (manual: $\mu = 31\%$; mixed: $\mu = 49\%$; auto: $\mu = 47\%$). This means that on average, manual users waste 1 iteration, mixed users waste 41 iterations, and auto users waste 1026 iterations.

However, automated and mixed sequences result in a 3% higher p_S compared to manual sequences for the same task.This is likely due to a greater coverage of search space from the mixed and automated sequences compared to manual. There is a delicate balance between iterating in an efficient manner but exploring enough to achieve better results. We now present our approach to quantitatively estimating a sequence's breadth of exploration.

2.2.4.2 On Exploration

Exploring more model and preprocessing operators leads to higher p_S in manual sequences but does not improve p_S in automated sequences. Users iterating with manual sequences cover the same number of model types as automated sequences, but a lower variety of preprocessing techniques.

As shown in Figure 2.6, manual sequences explore a similar number of models as mixed and automated sequences (manual: $\mu = 3.08, \sigma = 1.63$; mixed: $\mu = 2.64, \sigma = 1.37$; auto: $\mu = 3.31, \sigma = 1.48$). However, there tends to be less manual exploration on data preprocessing when compared to the mixed and automated groups (manual: $\mu = 1.51, \sigma = 1.82$; mixed: $\mu = 2.49, \sigma = 1.44$; auto: $\mu = 3.22, \sigma = 1.91$). It is interesting to note that automated sequences explore the same number of preprocessing operators as models on average, while manual sequences have a higher tendency to completely leave out data preprocessing from their workflows.

[6]Sequences with a length of 1 were excluded from this and other appropriate analyses in this section to avoid inflation by deltas of 0, counts of 1, or percentages of 100.

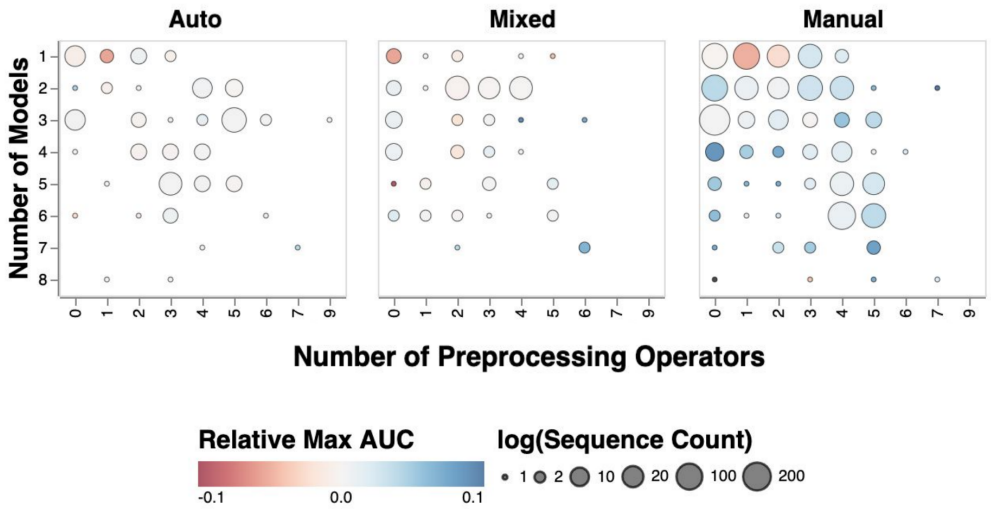


Figure 2.7: Joint distribution of model and preprocessing operators per sequence. Max AUC is relative to only the sequences within each category.

When examining the number of model and preprocessing operators jointly, as shown in Figure 2.7, we can see that for manual iterations, trying more combinations leads to better performance, but this is not the case in mixed and auto sequences. For each of the combinations that auto sequences explore, they typically perform many more iterations of hyperparameter tuning than manual sequences do for a given combination, evident from the fact that most auto iteration sequences comprise of hyperparameter tuning, as shown in Table 2.6, and that auto sequences are much longer. These trends reveal two phenomena: 1) when a combination is explored using default or rule-of-thumb hyperparameters in just a few iterations, as is the case in typical manual sequences, the performance gap between different combinations is large; 2) when a combination is explored with many hyperparameter tuning iterations, as is the case in typical auto sequences, the performance gap shrinks between different combinations, leading to diminishing returns from exploring more combinations on p_S improvement. In other words, auto sequences often waste most iterations on hyperparameter tuning to improve the performance of suboptimal combinations without improving p_S, while the potential of a combination can be estimated quickly with just a few iterations. However, there is merit to extensive hyperparameter tuning, evident in the fact that auto sequences achieve a 3% higher p_S than manual ones as discussed in Section 2.2.4.1. Together, these insights suggest that a hybrid approach, wherein coarse-grained search is first performed over the combinations of preprocessing and model operators, followed by fine-grained search doing hyperparameter tuning on the most promising combinations, would be both effective and also efficient at finding a high-performing workflow.

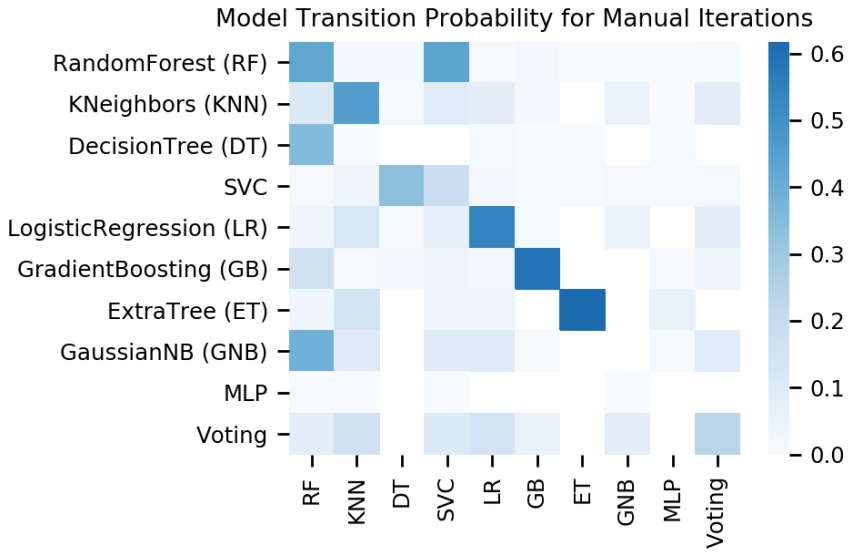


Figure 2.8: Model transition likelihood in consecutive manual iterations (row represents current iteration, and column represents next iteration).

Model	Manual	Mixed	Auto
RandomForest	40.83%	76.90%	53.32%
KNeighbors	37.99%	37.74%	50.54%
SVC	28.25%	23.73%	46.26%

Table 2.7: Mean % iterations per sequence for the top 3 models.

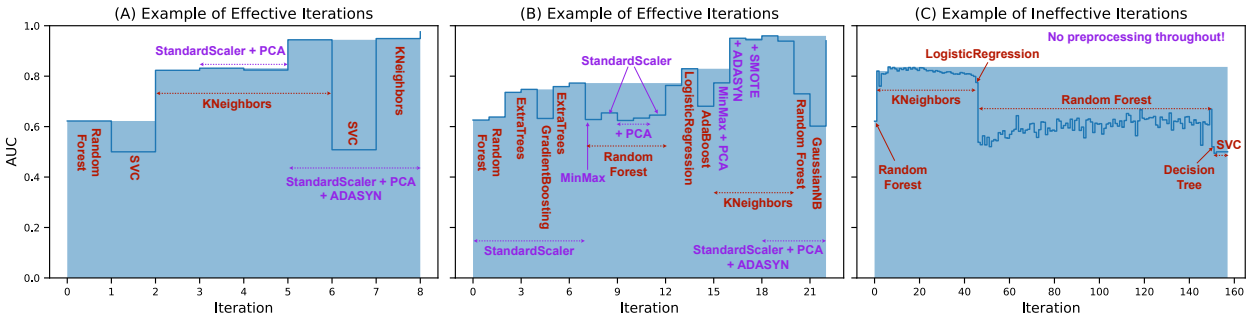


Figure 2.9: Examples of effective and ineffective sequences for a popular classification task. The shaded blue area represents the maximum AUC up until that point, the red text describes model changes, and the purple text describes preprocessing changes.

2.2.4.3 On Model Tuning

While automated sequences are dominated by hyperparameter changes, users iterating with manual sequences focus on model selection and eventually converge on high-performing models.

Since changing the ML model is the most common type of manual workflow change, making up 45.04% of all manual runs as shown in Table 2.6, we look into which models are abandoned and which are kept from one iteration to the next. Table 2.7 shows that in all groups (manual, mixed, and auto), users are much less likely to stick with SVC thank RF and KNN. Whenever SVC is used in a manual sequence, it accounts for only 28.25% of the runs, compared to 40.83% for RF and 37.99% for KNN. Combined with our findings from Figure 2.4 that RF and KNN perform better than SVC on average, we conclude that users are able to waste fewer iterations on models that do not work as well.

Furthermore, the transition probabilities between different pairs of models in Figure 2.8 reveal that users tend to stay with the same model from one iteration to the next, as evidenced by the high probabilities along the diagonal of the matrix. However, when a switch occurs, the model that is switched to the most is RF, with 26.22% of all model changes transitioning to RF.

2.2.5 Case Studies

So far, we have described insights aggregated across multiple users, reflecting the population-level trends in ML development. In this section, we dive deep into a few examples of both *effective* and *ineffective* practices to offer a complementary view on user behavior. We define a highly effective sequence as one that achieves a high AUC score over just a few iterations, wasting few to no iterations after p_S has been achieved.

In Figure 2.9, we show examples of highly-effective sequences (A and B), and an ineffective sequence (C) for the supervised classification task that was attempted by the highest number of users on OpenML. The dataset used in this binary classification task most notably has a very imbalanced class distribution, with 98.35% of the instances belonging to the majority class. Workflows that account for the class imbalance problem are able to achieve higher performance than those who do not.

Figure 2.9(A) illustrates one such example of a user who was successful in creating a near-optimal ML workflow for this task. This user iterated on the workflow using a *manual* sequence (based on our definition in Section 2.2.4), starting off by selecting a model, then adding data preprocessing, beginning with StandardScaler and Principal Component Analysis (PCA). They then incorporated an oversampling strategy, ADASYN, to counter the the class imbalance problem, resulting in a significant performance boost.

Similarly, Figure 2.9(B) also illustrates a user who iterated with a manual sequence, experimenting with different combinations of model and preprocessing operators, before discovering the high impact of data oversampling. As soon as they incorporated ADASYN,

they were able to quickly run through a final model selection phase to achieve one of the highest AUC scores for the task.

Finally, as shown in Figure 2.9(C), the user was off to a strong start by picking similar models as chosen in (A) and (B). However, the user then spent over one hundred *automated* iterations exhaustively tuning hyperparameters, resulting in over 95% of their iterations being wasted, i.e., they did not improve p_S. This user could benefit from learning the strategy adopted in manual approaches in (A) and (B) to first examine dataset characteristics to guide modeling decisions, instead of optimizing hyperparameters prematurely.

2.3 Implications for System Design

The results from the two studies demonstrate that ML model development is a highly iterative, human-in-the-loop process. The qualitative and quantitative conclusions drawn from these findings have many profound implications on designing systems to support human-in-the-loop ML model development.

The results in Section 2.1 suggest a number of properties that a versatile and effective human-in-the-loop ML system should possess:

- **Iteration.** Developers iterate on their workflows in every application domain and test out changes to all components of the workflow. Understanding the most frequent changes helps us develop systems that anticipate and respond rapidly to iterative changes.

- **Fine-grained feature engineering.** Handcrafted features designed using domain knowledge are still an indispensable part of the workflow development systems in all domains and should therefore be adequately supported instead of dismissed as an outdated practice.

- **Efficient joins.** Data is often pooled from multiple sources, thus requiring systems to support efficient joins in the data pre-processing component.

- **Explainable models.** Many domains have yet to embrace deep learning due to their needs for explainable models. The system should provide ample support to help developer interpret model behaviors.

- **Fast model training.** The fact that the most tuned model parameters are related to training time suggests that developers are in need of systems that have fast model training, but also low latency for the end-to-end workflow execution in general.

- **Fine-grained results analysis.** Fine-grained and summary evaluation methods are equally popular across all domains. Thus, model management systems should provide support for not only summary metrics but also more detailed model characteristics.

On the other hand, the findings in the second study involving analyzing over 475k user-generated runs on OpenML unveil more fine-grained user behaviors. We find that users often adopt a manual, automated, or mixed approach when iterating on their workflows, leading to varying level of success, potentially due to the expertise of the user. Patterns in these behaviors suggest a number of opportunities to assist and automate model development, especially towards the goal of empowering novice users.

- **Operator Recommendation.** Run-level statistics in Section 2.2.3 illustrate the prevalence of different operators and whether or not users are able to effectively use them. The case study in Section 2.2.3.1 shows that there is a discrepancy between the popularity and efficacy of the different operators. A human-in-the-loop ML system [110] could surface lesser-known but high-potential operators to educate the users and bridge the gap between the system's capabilities and the user's knowledge on specific capabilities.

- **Knowledge Transfer.** In a similar vein, the case studies in Section 2.2.5 demonstrate another opportunity for a human-in-the-loop ML system to assist in model development by amassing crowdsourced best practices to guide novices to more effectively iterate on the workflow.

- **Balance between Manual Development and Automation.** We observe that manual approaches result in fewer wasted iterations compared to automated approaches. Yet, automated approaches often involve more preprocessing and hyperparameter options explored, resulting in higher performance overall—suggesting potential benefits for a human-in-the-loop ML system that appropriately recommends a clever combination of the two strategies.

Next, we present a declarative system aimed at accelerating iteration in human-in-the-loop model development by optimizing the end-to-end ML workflow in Chapter 3, using many of the findings listed above as guiding principles and for evaluation on realistic workloads.

Chapter 3

Accelerating Model Development with Helix

As shown in Chapter 2, machine learning workflow development is a process of trial-and-error: developers iterate on the machine learning workflow by incrementally modifying steps within, including (i) *preprocessing*: altering data cleaning or extraction, or engineering features; (ii) *model training*: tweaking hyperparameters, or changing the objective or learning algorithm; and (iii) *postprocessing*: evaluating with new data, or generating additional statistics or visualizations. These iterations are necessitated by the difficulties in predicting the performance of a workflow *a priori*, due to both the variability of data and the complexity and unpredictability of machine learning.

Prior to this work, the majority of effort in building better tools to support machine learning has focused on speeding up the model training process [27, 1, 181, 200, 179], which, given the knowledge gleaned from the studies in Chapter 2, is only a small piece of the model development process. In this chapter, we present Helix, a system that optimizes for the iterative model development process by intelligently caching and reusing, or recomputing intermediate results in the end-to-end ML workflow across iterations.

During development, users tend to rerun the entire ML workflow from scratch in every iteration, due to the fact that it is common for users to develop the entire workflow in a single script, which lends itself to end-to-end execution. However, as we saw in Chapter 2, the changes across iterations are *incremental*, which suggests that most of the recomputation in each iteration is in fact redundant. Given that each iteration can easily take hours to execute, there is an opportunity here to eliminate a great deal of inefficiency.

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

Unfortunately, existing machine learning systems do not optimize for rapid iteration. For example, KeystoneML [177], 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 [221], targeted at knowledge-base construction, materializes the results of all of the feature extraction and engineering steps, while also applying approximate inference to speed up model training. Although this naïve materialization approach does lead to reuse in iterative executions, it is wasteful and time-consuming.

We present HELIX, a *declarative, general-purpose machine learning system that optimizes across iterations*. HELIX is able to match or exceed the performance of KeystoneML and DeepDive on one-shot execution, while achieving up to 95% computation time reduction on iterative execution across four real-world applications. By optimizing across iterations, HELIX allows data scientists to avoid wasting time running the workflow from scratch every time they make a change and instead run their workflows in time proportional to the complexity of the change made. HELIX is able to thereby substantially increase developer productivity while simultaneously lowering resource consumption.

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

Challenges in Iterative Execution Optimization. ML workflows in practice can be quite large and complex. As we described previously, one simple approach to enable iterative execution optimization (adopted by DeepDive) is to materialize every single node, such that the next time the workflow is run, we can simply check if the result can be reused from the previous iteration, and if so, reuse it. Unfortunately, this approach is not only wasteful in storage but also potentially very time-consuming due to materialization overhead. Moreover, in a subsequent iteration, it may be cheaper to recompute an intermediate result, as opposed to reading it from disk.

A better approach is to determine whether a node is worth materializing by considering both the time taken for computing a node and the time taken for computing its ancestors. Then, during subsequent iterations, we can determine whether to read the result for a node from persistent storage (if materialized), which could lead to large portions of the graph being pruned, or to compute it from scratch. In this paper, we prove that the reuse plan problem is in PTIME via a non-trivial ***reduction to* Max-Flow *using the* Project Selection Problem *[95], while the materialization problem is, in fact,* NP-Hard**.

Challenges in Specification and Generalization. To enable iterative execution optimization, we need to support the specification of the end-to-end machine learning workflow in a high-level language. This is challenging because data preprocessing can vary greatly across applications, often requiring ad-hoc code involving complex composition of declarative statements and UDFs [15], making it hard to automatically analyze the workflow to apply holistic iterative execution optimization.

We adopt a hybrid approach within HELIX: developers specify their workflow in an ***intuitive, high-level domain-specific language (DSL) in Scala*** (similar to existing

systems like KeystoneML), using ***imperative code as needed for UDFs***, say for feature engineering. This interoperability allows developers to seamlessly integrate existing JVM machine learning libraries [49, 161]. 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 [207].

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 [221], HELIX is not restricted to regression or factor graphs, allowing data scientists to use the most suitable model for their tasks. All of the functions in Scikit-learn's (a popular ML toolkit) can be mapped to functions in the DSL [208], allowing HELIX to easily capture applications ranging from natural language processing, to knowledge extraction, to computer vision. Moreover, by studying the variation in the dataflow graph across iterations, HELIX is able to identify reuse opportunities across iterations. Our work is a first step in a broader agenda to improve human-in-the-loop ML [206].

Contributions and Outline. The rest of the chapter is organized as follows: Section 3.1 presents an architectural overview of the system, and a concrete workflow to illustrate concepts discussed in the subsequent sections; Section 3.2 describes the programming interface for effortless end-to-end workflow specification; Section 3.3 discusses HELIX system internals, including the workflow DAG generation and change tracking between iterations; Section 3.4 formally presents the two major optimization problems in accelerating iterative ML and HELIX's solution to both problems. We evaluate our framework on four workflows from different applications domains and against two state-of-the-art systems in Section 3.5. We discuss related work in Section 3.6.

3.1 Background and Overview

In this section, we describe the HELIX system architecture and present a sample workflow in HELIX that will serve as a running example.

3.1.1 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 3.1. The three components work collectively to *minimize the execution time for both the current iteration and subsequent iterations*:

1. Programming Interface (Section 3.2). HELIX provides a single Scala interface named `Workflow` for programming the entire workflow; the HELIX DSL also enables embedding of imperative code in declarative statements. Through just a handful of extensible operator types, the DSL supports a wide range of use cases for both data preprocessing and machine learning.

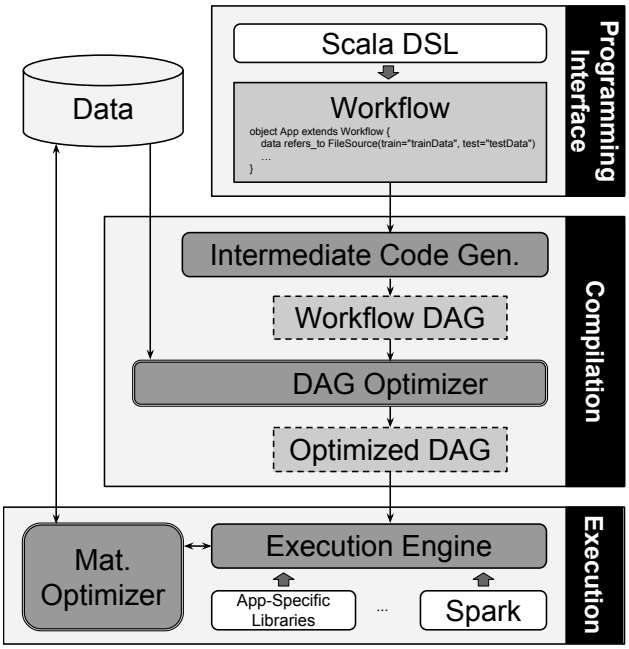


Figure 3.1: HELIX System architecture. A program written by the user in the HELIX DSL, known as a `Workflow`, is first compiled into an intermediate DAG representation, which is optimized to produce a physical plan to be run by the execution engine. At runtime, the execution engine selectively materializes intermediate results to disk.

2. Compilation (Sections 3.3, 3.4.1–3.4.2). A `Workflow` is internally represented as a directed acyclic graph (DAG) of operator outputs. The DAG is compared to the one in previous iterations to determine reusability (Section 3.3). The *DAG Optimizer* uses this information to produce an optimal *physical execution plan* that *minimizes the one-shot runtime of the workflow*, by selectively loading previous results via a MAX-FLOW-based algorithm (Section 3.4.1–3.4.2).

3. Execution Engine (Section 3.4.3). The execution engine carries out the physical plan produced during the compilation phase, while communicating with the *materialization operator* to materialize intermediate results, to *minimize runtime of future executions*. The execution engine uses Spark [219] for data processing and domain-specific libraries such as CoreNLP [120] and Deeplearning4j [49] for custom needs. HELIX defers operator pipelining and scheduling for asynchronous execution to Spark. Operators that can run concurrently are invoked in an arbitrary order, executed by Spark via Fair Scheduling. While by default we use Spark in the batch processing mode, it can be configured to perform stream processing using the same APIs as batch. We discuss optimizations for streaming in Section 3.4.

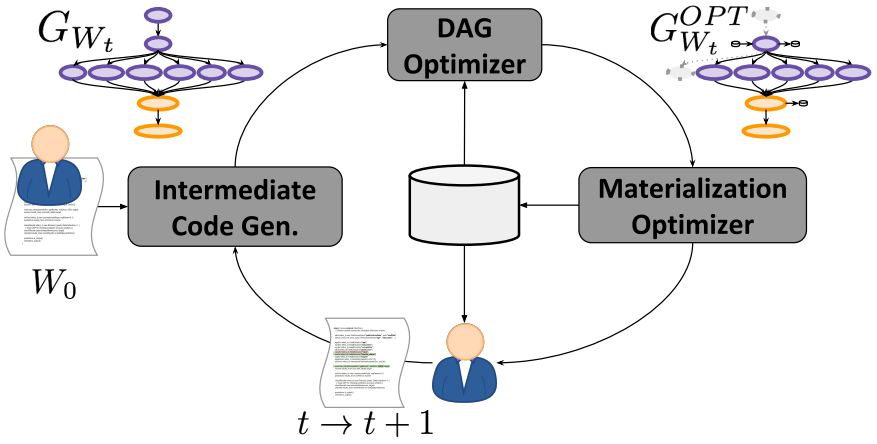


Figure 3.2: Roles of system components in the HELIX workflow lifecycle.

3.1.2 The Workflow Lifecycle

Given the system components described in the previous section, Figure 3.2 illustrates how they fit into the lifecycle of ML workflows introduced in Section 1.2. Starting with W_0, an initial version of the workflow, the lifecycle includes the following stages:

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

Without loss of generality, we assume that a workflow W_t is only executed once in each iteration. We model a repeated execution of W_t as a new iteration where $W_{t+1} = W_t$. Distinguishing two executions of the same workflow is important because they may have different run times—the second execution can reuse results materialized in the first execution for a potential run time reduction.

3.1.3 Example Workflow

We demonstrate the usage of HELIX with a simple example ML workflow for predicting income using census data from Kohavi [98], shown in Figure 3.3a); 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

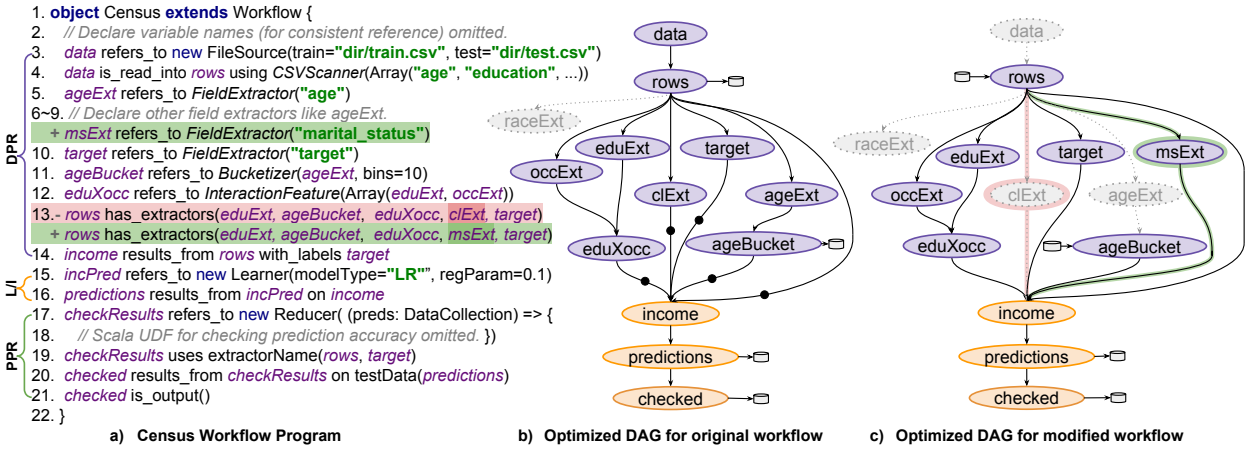


Figure 3.3: Example workflow for predicting income from census data.

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 `occExt`.

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` 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 3.3a) into a DAG, which contains all nodes including `raceExt` and all edges (including the dashed edge) except the ones marked with dots in Figure 3.3b). 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 `msExt` is added (below line 9), and `clExt` is removed (line 13); correspondingly, in the updated DAG, a new node is added for `msExt` (green edges), while `clExt` gets pruned (pink edges). In addition, HELIX chooses to load materialized results for `rows` from the previous iteration allowing `data` to be pruned, avoiding a costly parsing step. HELIX also loads `ageBucket` instead of recomputing the bucket boundaries requiring a full scan. HELIX materializes `predictions` in both iterations since it has changed. Although `predictions` is not reused in the updated workflow, its materialization has high expected payoff over iterations because PPR iterations (changes to `checked` in this case) are the most common as per our applied ML literature survey results shown in Figure 2.3. This 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.

3.2 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 [125], 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 [167].

3.2.1 Operations in ML Workflows

In this section, we argue that common operations in ML workflows can be decomposed into a small set of *basis functions* $\mathcal{F}$. We first introduce $\mathcal{F}$ and then enumerate its mapping

onto operations in Scikit-learn [151], one of the most comprehensive ML libraries, thereby demonstrating coverage. In Section 3.2.2, we introduce HML, which implements the capabilities offered by $\mathcal{F}$.

As mentioned in Section 3.1, an ML workflow consists of three components: data preprocessing (DPR), learning/inference (L/I), and postprocessing (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 MLLib [126], TFX [22], 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), $\mathbf{x} \in \mathbb{R}^d$. For notational convenience, we denote a d-dimensional FV, $\mathbf{x} \in \mathbb{R}^d$, as $\mathbf{x}^d$. 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 $\mathcal{D}$.

DPR. The goal of DPR is to transform raw input data into $\mathcal{D}$. 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 $\mathcal{S} = \{r\}$ 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* $r \mapsto \mathbf{x}^d$: extracting features from a record.
- *Feature Transformation* $T : \mathbf{x}^d \mapsto \mathbf{x}^{d'}$: deriving a new set of features from the input features.
- *Feature Concatenation* $(\mathbf{x}^{d_1}, \mathbf{x}^{d_2}, \ldots) \mapsto \mathbf{x}^{\sum_i d_i}$: 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 $\mathcal{D}$. We address this use case along with L/I next.

L/I. At a high-level, L/I is about learning a function f from the input $\mathcal{D}$, where $f : \mathcal{X} \to \mathbb{R}^{d'}, d' \geq 1$. 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 $\mathcal{D}$, 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 [90]. Thus, L/I can be decomposed into:

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

Note that labels can be represented as FVs like other features, hence the usage of a single $\mathcal{D}$ 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, data visualization, and other application-specific activities. 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 $\mathcal{D}$. We refer to a computation with output sizes independent of input sizes as a *reduce:*

- *Reduce* $(\mathcal{D}, s') \mapsto s$: 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 *reduce* optimizes, learning various models and outputting s, which can represent a function corresponding to the model with the best cross-validated hyperparameters.

3.2.1.1 Comparison with Scikit-learn

A dataset in Scikit-learn is represented as a matrix of FVs, denoted by `X`. This is conceptually equivalent to $\mathcal{D} = \{\mathbf{x}^d\}$ 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 [174]. Operations like loading and transformations that do not tailor their behavior to particular characteristics present in the dataset $\mathcal{D}$ map trivially onto the DPR basis functions $\in \mathcal{F}$ introduced at the start of Section 3.2.1, so we focus on comparing data-dependent DPR and L/I, and model selection and evaluation.

Scikit-learn Operations for DPR and L/I. Scikit-learn objects for DPR and L/I implement one or more of the following interfaces [31]:

- **Estimator**, used to indicate that an operation has data-dependent behavior via a `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 `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 `predict_proba`.
- **Transformer**, used to indicate that the operation may be used for feature transformations via a `transform(X)` method, taking a matrix of FVs and producing a new matrix $\mathtt{X}_{new}$.

Scikit-learn DPR, L/I	Composed Members of $\mathcal{F}$
`fit(X[, y])`	*learning* ($\mathcal{D} \mapsto f$)
`predict_proba(X)`	*inference* ($(\mathcal{D}, f) \mapsto \mathcal{Y}$)
`predict(X)`	*inference*, optionally followed by *transformation*
`fit_predict(X[, y])`	*learning*, then *inference*
`transform(X)`	*transformation* or *inference*, depending on whether operation is learned via prior call to `fit`
`fit_transform(X)`	*learning*, then *inference*
Scikit-learn PPR	**Composed Members of $\mathcal{F}$**
eval: `score(`$\mathtt{y}_{true}$`,` $\mathtt{y}_{pred}$`)`	*join* $\mathtt{y}_{true}$ and $\mathtt{y}_{pred}$ into a single dataset $\mathcal{D}$, then *reduce*
eval: `score(op, X, y)`	*inference*, then *join*, then *reduce*
selection: `fit(`$p_1, \ldots, p_n$`)`	*reduce*, implemented in terms of *learning*, *inference*, and *reduce* (for scoring)

Table 3.1: Scikit-learn DPR, L/I, and PPR coverage in terms of $\mathcal{F}$.

An operation implementing both Estimator and Predictor has a `fit_predict` method, and an operation implementing both Estimator and Transformer has a `fit_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 `fit`. For example, a useful data-dependent feature transformation for a Naive 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 3.1 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 separate interfaces for inference (via Predictor) and data-dependent transformations (via Transformer); the separation exists mainly to draw attention to the semantic separation between DPR and L/I.

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 `fit` and `score`. The `fit` method takes a set of hyperparameters over which to search, with different models scored according to the `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 3.1, Scikit-learn's operations for evaluation may be implemented via compositions of (optionally) *inference*, *joining*, 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.

3.2.2 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 3.3a). The complete grammar for HML in Backus-Naur Form can be found in Figure 3.4 and the semantics of all of the expressions can be found in Table 3.2. Here, we describe high-level concepts including DCs and operators and discuss the strengths and limitations of HML in Section 3.2.3.

3.2.2.1 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 3.3a), or from operations on other DCs, e.g., `rows` in Line 4 in Figure 3.3a). An element in a DC can either be a *semantic unit*, the data structure for DPR, or an *example*, the data structure for L/I.

A DC can only contain a single type of element. DC_{SU} and DC_E denote a DC of semantic units and a DC of examples, respectively. The type of elements in a DC is determined by the operator that produced the DC and not explicitly specified by the user. We elaborate on the relationship between operators and element types in Section 3.2.2.2, 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 $\mathcal{D}$ to learn each operator separately can be highly inefficient. We introduce the notion of semantic units (SU) to compartmentalize the logical and physical

representations of features, so that the learning of DPR functions can be delayed and batched.

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

Examples. Examples gather all the FVs contained in the output of various SUs into a single FV for learning. Formally, an example contains a set of SUs S, and an optional pointer to one of the SUs whose output will be used as the label in supervised settings, and an output FV, which is formed by concatenating the outputs of S. In the implementation, the order of SUs in the concatenation is determined globally across $\mathcal{D}$, and SUs whose outputs are not FVs are filtered out.

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

Unified learning support. HML provides unified support for training and test data by treating them as a single DC, as done in Line 4 in Figure 3.3a). This design ensures that both training and test data undergo the exact same data preprocessing steps, eliminating bugs caused by inconsistent data preprocessing procedures handling training and test data separately. Helix automatically selects the appropriate data for training and evaluation. However, if desired, users can handle training and test data differently by specifying separate DAGs for training and testing. Common operators can be shared across the two DAGs without code duplication.

3.2.2.2 Operators

Operators in Helix are designed to cover the functions enumerated in Section 3.2.1, using the data structures introduced above. A Helix *operator* takes one or more DCs and outputs DCs, ML models, or scalars. Each operator encapsulates a function f, written in Scala, to be applied to individual elements in the input DCs. 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 $\mathcal{F}$ enumerated in Section 3.2.1 below.

Scanner. *Scanner* is the interface for parsing $\in \mathcal{F}$ and acts like a flatMap, i.e., for each input element, it adds zero or more elements to the output DC. Thus, it can also be used to perform filtering. The input and output of Scanner are DC_{SU}s. `CSVScanner` in Line 4 of Figure 3.3a) is an example of a Scanner that parses lines in a CSV file into key-value pairs for columns.

Synthesizer. *Synthesizer* supports join $\in \mathcal{F}$, for elements both across multiple DCs and within the same DC. Thus, it can also support aggregation operations such as sliding windows in time series. Synthesizers also serve the important purpose of specifying the set of SUs that make up an example (where output FVs from the SUs are automatically assembled into a single FV). In the simple case where each SU in a DC_{SU} corresponds to an example, a pass-through synthesizer is implicitly declared by naming the output DC_E, such as in Line 14 of Figure 3.3a).

Learner. *Learner* is the interface for learning and inference $\in \mathcal{F}$, in a single operator. A learner operator L contains a learned function f, which can be 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_E. For example, the learner `incPred` in Line 15 of Figure 3.3a) is a learner trained on the "train" portion of the DC_E `income` and outputs inference results as the DC_E `predictions`.

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

Reducer. Reducer is the interface for reduce $\in \mathcal{F}$ and thus the main operator interface for PPR. The inputs to a reducer are DC_E and an optional scalar and the output is a scalar, where scalars refer to non-dataset objects. For example, `checkResults` in Figure 3.3a) Line 17 is a reducer that computes the prediction accuracy of the inference results in `predictions`.

3.2.3 Scope and Limitations

Coverage. In Section 3.2.1, we described how the set of basis operations $\mathcal{F}$ we propose covers all major operations in Scikit-learn, one of the most comprehensive ML libraries. We then showed in Section 3.2.2 that HML captures all functions in $\mathcal{F}$. 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 [126] and Weka [73], 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 MLLib models implement the train (equivalent to learning) and predict (equivalent to inference) methods, they can easily be plugged into

Phrase	Usage	Operation	Example
refers_to	*string* refers_to HELIX *object*	Register a HELIX *object* to a *string* name	"ext1" refers_to Extractor(...)
is_read_into ... using	$DC_i[SU]$ is_read_into $DC_j[SU]$ using *scanner*	Apply *scanner* on DC_i to obtain DC_j	"sentence" is_read_into "word" using whitespaceTokenizer
has_extractors	$DC[SU]$ has_extractors *extractor+*	Apply extractors to DC	"word" has_extractors ("ext1", "ext2")
on	*synthesizer/learner/reducer* on $DC[*]+$	Apply *synthesizer/learner* on input DC(s) to produce an output DC[E]	"match" on ("person_candidate" , "known_persons")
results_from	$DC_i[E]$ results_from $DC_j[*]$ [with_label *extractor*]	Wrap each element in DC_i in an Example and optionally labels the Examples with the output of *extractor*.	"income" results_from "rows" with_label "target"
	$DC[E]$/Scalar results_from *clause*	Specify the name for *clause*'s output DC[E].	"learned" results_from "L" on "income"
uses	*synthesizer/learner/reducer* uses *extractors+*	Specify *synthesizer/learner*'s dependency on the output of *extractors+* to prevent pruning or uncaching of intermediate results due to optimization.	"match" uses ("ext1", "ext2")
is_output	$DC[*]$/*result* is_output	Requires *DC/result* to be materialized.	"learned" is_output

Table 3.2: Usage and functions of key phrases in HML. $DC[A]$ denotes a DC with name DC and elements of type $A \in \{SU, E\}$, with $A = *$ indicating both types are legal. $x+$ indicates that x appears one or more times. When appearing in the same statement, on takes precedence over results_from.

⟨*var*⟩ ::= ⟨*string*⟩
⟨*scanner*⟩ ::= ⟨*var*⟩ | ⟨*scanner-obj*⟩
⟨*extractor*⟩ ::= ⟨*var*⟩ | ⟨*extractor-obj*⟩
⟨*typed-ext*⟩ ::= '(' ⟨*var*⟩ ',' ⟨*extractor*⟩ ')'
⟨*extractors*⟩ ::= '(' ⟨*extractor*⟩ { ',' ⟨*extractor*⟩ } ')'
⟨*typed-exts*⟩ ::= '(' ⟨*typed-ext*⟩ {',' ⟨*typed-ext*⟩} ')'
⟨*obj*⟩ ::= ⟨*data-source*⟩ | ⟨*scanner-obj*⟩ | ⟨*extractor-obj*⟩ | ⟨*learner-obj*⟩ | ⟨*synthesizer-obj*⟩ | ⟨*reducer-obj*⟩
⟨*assign*⟩ ::= ⟨*var*⟩ '`refers_to`' ⟨*obj*⟩
⟨*expr1*⟩ ::= ⟨*var*⟩ '`is_read_into`' ⟨*var*⟩ '`using`' ⟨*scanner*⟩
⟨*expr2*⟩ ::= ⟨*var*⟩ '`has_extractors`' ⟨*extractors*⟩
⟨*list*⟩ ::= ⟨*var*⟩ | '(' ⟨*var*⟩ ',' ⟨*var*⟩ { ',' ⟨*var*⟩ } ')'
⟨*apply*⟩ ::= ⟨*var*⟩ '`on`' ⟨*list*⟩
⟨*expr3*⟩ ::= ⟨*apply*⟩ '`as_examples`' ⟨*var*⟩
⟨*expr4*⟩ ::= ⟨*apply*⟩ '`as_results`' ⟨*var*⟩
⟨*expr5*⟩ ::= ⟨*var*⟩ '`as_examples`' ⟨*var*⟩
'`with_labels`' ⟨*extractor*⟩
⟨*expr6*⟩ ::= ⟨*var*⟩ '`uses`' ⟨*typed-exts*⟩
⟨*expr7*⟩ ::= ⟨*var*⟩ '`is_output()`'
⟨*statement*⟩ ::= ⟨*assign*⟩ | ⟨*expr1*⟩ | ⟨*expr2*⟩ | ⟨*expr3*⟩ | ⟨*expr4*⟩ | ⟨*expr5*⟩ | ⟨*expr6*⟩ | ⟨*expr7*⟩ | ⟨*Scala expr*⟩
⟨*program*⟩ ::= '`object`' ⟨*string*⟩ '`extends Workflow {`'
{ ⟨*statement*⟩ ⟨*line-break*⟩ }
'`}`'

Figure 3.4: HML 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 HML and Scala expressions, a direct benefit of being an embedded DSL.

Learner in HELIX. We demonstrate in Section 3.5 that the current set of implemented operations is sufficient for supporting applications across different domains.

Limitations. Since HELIX currently relies on its Scala DSL for workflow specification, popular non-JVM libraries, such as TensorFlow [1] and Pytorch [148], 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 HELIX operators and those in standard ML libraries, as established in Section 3.2.2. That said, our contributions in materialization and reuse apply across all languages. In the future, we plan on abstracting the DAG representation in HELIX into a language-agnostic system that can sit below the language layer for all DAG based systems, including TensorFlow and Spark.

The other downside of HML is that ML models are treated largely as black boxes. Thus, work on optimizing learning, e.g., [163, 224], is orthogonal to (and can therefore be combined

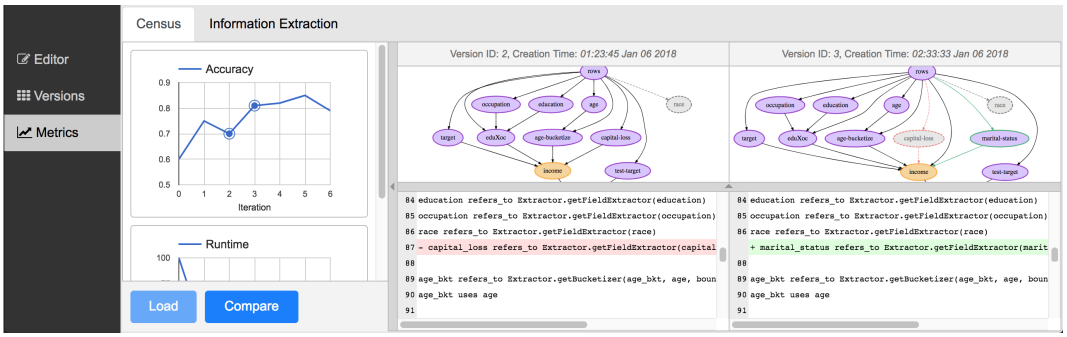


Figure 3.5: Metrics aggregation and version comparison in the HELIX IDE. Users can select and compare specific versions, represented as points on the metric trend lines, for code change and visualized execution plans, in order to better understand the performance impact of specific modifications.

with) our work, which operates at a coarser granularity.

3.2.4 Integrated Development Environment

To demonstrate the full range of capabilities in HELIX, we implemented a prototype integrated development environment (IDE) that provides versioning and metrics tracking capabilities on top of the DSL. The IDE comprises three sections that the user can toggle between to perform different tasks: code editor, versions, metrics.

Code Editor. The HELIX IDE provides HELIX DSL specific autocomplete and syntax highlighting to facilitate programming. A "Suggest Modifications" button lets user request machine-generated edits to be shown inline using Github-style code change highlighting, as illustrated in Figure 3.3a), thus allowing users to iterate rapidly on the workflow without mastering the DSL. Once the workflow is executed, the user will be able to inspect the optimized execution plan in the DAG format, as shown in Figure 3.3b). Individual runtime and storage for each operation are displayed by hovering over them.

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

Metrics. As shown in Figure 3.5, the Metrics tab aggregates the evaluation metrics for the workflow across iterations into plots with the metric value on the y-axis and the iteration number on the x-axis. Each point in the plot represents a version of the workflow. Users can select a single point to load the associated code version or two points for comparison. In Figure 3.5, Version 2 and 3 are selected in the Accuracy plot for comparison. The comparative view visualizes the DAG and highlights changes in the DAG using git-like visual comparison cues, in addition to showing the two versions of the workflow code also with changes highlighted. This feature enables rapid exploration of the relationships between var-

ious metrics and changes to specific components of the workflow. Understanding the impact of each past iteration is crucial for making effective future improvements, thus reducing the overall number of iterations to achieve the desired outcome.

3.3 Compilation and Representation

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

3.3.1 The Workflow DAG

At compile time, HELIX'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* HELIX *operators $F = \{f_i\}$, the Workflow DAG is a directed acyclic graph $G_W = (N, E)$, where node $n_i \in N$ represents the output of $f_i \in F$ and $(n_i, n_j) \in E$ if the output of f_i is an input to f_j.*

Figure 3.3b) shows the Workflow DAG for the program in Figure 3.3a). Nodes for operators involved in DPR are colored purple whereas those involved in L/I 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 intermediate code generator introduces edges not specified in the Workflow between the extractor and the synthesizer nodes, such as the edges marked by dots ($\bullet$) in Figure 3.3b). These edges connect extractors to downstream DCs in order to automatically aggregate all features for learning. One concern is that this may lead to redundant computation of unused features; we describe pruning mechanisms to address this issue in Section 3.4.4.

3.3.2 Tracking Changes

As described in Section 3.1.2, a user starts with an initial workflow W_0 and iterates on this workflow. Let W_t be the version of the workflow at iteration $t \geq 0$ with the corresponding DAG $G_W^t = (N_t, E_t)$; W_{t+1} denotes the workflow obtained in the next iteration. To describe the changes between W_t and W_{t+1}, we introduce the notion of *equivalence*.

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

Equivalence is symmetric, i.e., $n_i^{t'} \equiv n_i^t \Leftrightarrow n_i^t \equiv n_i^{t'}$, and transitive, i.e., $(n_i^t \equiv n_i^{t'} \wedge n_i^{t'} \equiv n_i^{t''}) \Rightarrow n_i^t \equiv n_i^{t''}$. Newly added operators in W_{t+1} do not have equivalent nodes in W_t;

neither do nodes in W_t that are removed in W_{t+1}. For a node that persists across iterations, we need both the operator and the ancestor nodes to stay the same for equivalence. Using this definition of equivalence, we determine if intermediate results on disk can be safely reused through the notion of equivalent materialization:

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

One challenge in determining equivalence is deciding whether two versions of an operator compute the same results on the same input. For arbitrary functions, this is undecidable as proven by Rice's Theorem [165]. The programming language community has a large body of work on verifying operational equivalence for specific classes of programs [204, 154, 69]. HELIX currently employs a simple representational equivalence verification—an operator remains equivalent across iterations if its declaration in the DSL is not modified and all of its ancestors are unchanged. Incorporating more advanced techniques for verifying equivalence is future work.

To guarantee correctness, i.e., results obtained at iteration t reflect the specification for W_t and are computed from the appropriate input, we impose the constraint:

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

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

Theorem 1. HELIX *returns the 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 given the results at t are correct, the results at iteration $t+1$ are incorrect, i.e., $\exists\ n_i^{t+1}$ s.t. the results for n_i^t are reused when n_i^{t+1} is original. Under the invariant conditions in Theorem 1, we can only have $n_i^{t+1} \not\equiv n_i^t$ 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_i^t for some original n_i^{t+1}. However, this violates Constraint 1. Therefore, the results for W_t are correct $\forall\ t \geq 0$. □

3.4 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 3.4.2), and OPT-MAT-PLAN, which decides what to materialize to accelerate future iterations (Section 3.4.3). We

finally discuss pruning optimizations to eliminate redundant computations (Section 3.4.4). We begin by introducing common notation and definitions.

3.4.1 Preliminaries

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

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

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

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

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

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

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

$$T(W, s) = \sum_{n_i \in N} \mathbb{I}\{s(n_i) = S_c\}\, c_i + \mathbb{I}\{s(n_i) = S_l\}\, l_i \tag{3.1}$$

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

Clearly, setting all nodes to S_p trivially minimizes Equation 3.1. However, recall that Constraint 1 requires all original operators to be rerun. Thus, if an original operator n_i is introduced, we must have $s(n_i) = S_c$, which by Constraint 2 requires that $S(n_j) \neq S_p\ \forall n_j \in parents(n_i)$. 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 3.1 next.

3.4.2 Optimal Execution Plan

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

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

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

$$T^*(W) = \min_s \ T(W, s) \tag{3.2}$$

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

Deciding to load certain nodes can have cascading effects since ancestors of a loaded node can potentially be pruned, leading to large reductions in run time. On the other hand, Constraint 2 disallows the parents of computed nodes to be pruned. Thus, the decisions to load a node n_i can be affected by nodes outside of the set of ancestors to n_i. For example, in the DAG on the left in Figure 3.6, loading n_7 allows n_{1-6} to be potentially pruned. However, the decision to compute n_8, possibly arising from the fact that $l_8 \gg c_8$, requires that n_5 must not be pruned.

Despite such complex dependencies between the decisions for individual nodes, Problem 1 can be solved optimally in polynomial time through a linear time reduction to the *project-selection problem* (PSP), which is an application of Max-Flow [95].

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

Reduction to the Project Selection Problem. We can reduce an instance of Problem 1 x to an equivalent instance of PSP y such that the optimal solution to y maps to an optimal solution of x. Let $G = (N, E)$ be the `Workflow` DAG in x, and P be the set of projects in y. We can visualize the prerequisite requirements in y as a DAG with the projects as the nodes and an edge (j, i) indicating that project i is a prerequisite of project j. The reduction, φ, depicted in Figure 3.6 for an example instance of x, is shown in Algorithm 1. For each node $n_i \in N$, we create two projects in PSP: a_i with profit $-l_i$ and b_i with profit $l_i - c_i$. We set a_i as the prerequisite for b_i. For an edge $(n_i, n_j) \in E$, we set the project a_i corresponding to node n_i as the prerequisite for the project b_j corresponding to node n_j. Selecting both projects a_i and b_i corresponding to n_i is equivalent to computing n_i, i.e., $s(n_i) = S_c$, while selecting only a_i is equivalent to loading n_i, i.e., $s(n_i) = S_l$. Nodes with neither projects selected are pruned. An example solution mapping from PSP to OEP is shown in Figure 3.6. Projects $a_4, a_5, a_6, b_6, a_7, b_7, a_8$ are selected, which cause nodes n_4, n_5, n_8 to be loaded, n_6 and n_7 to be computed, and n_1, n_2, n_3 to be pruned.

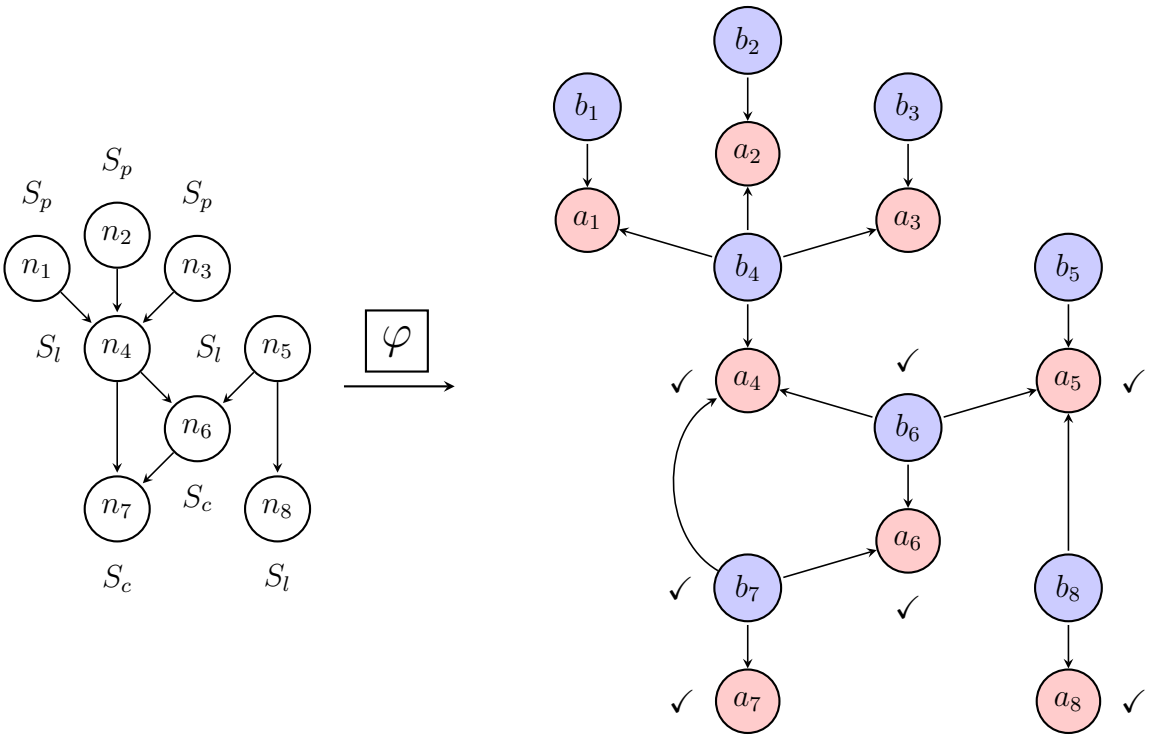


Figure 3.6: Transforming a `Workflow` DAG to a set of projects and dependencies. Checkmarks (✓) in the RHS DAG indicate a feasible solution to PSP, which maps onto the node states (S_p, S_c, S_l) in the LHS DAG.

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

Theorem 2. *Given an instance of* Opt-Exec-Plan *x, the reduction in Algorithm 1 produces a feasible and optimal solution to x.*

We first formulate Opt-Exec-Plan as an integer linear program (ILP) before presenting the proof itself. The proof depends on the ILP formulation to establish a mapping between

Algorithm 1: OEP via Reduction to PSP

Input: $G_W = (N, E), \{l_i\}, \{c_i\}$

$P \leftarrow \varnothing$;
for $n_i \in N$ **do**
  $P \leftarrow P \cup \{a_i\}$; // Create a project a_i
  $profit[a_i] \leftarrow -l_i$; // Set profit of a_i to $-l_i$
  $P \leftarrow P \cup \{b_i\}$; // Create a project b_i
  $profit[b_i] \leftarrow l_i - c_i$; // Set profit of b_i to $l_i - c_i$
  // Add a_i as prerequisite for b_i.;
  $prerequisite[b_i] \leftarrow prerequisite[b_i] \cup a_i$;
  for $(n_i, n_j) \in \{edges\ leaving\ from\ n_i\} \subseteq E$ **do**
    // Add a_i as prerequisite for b_j.;
    $prerequisite[b_j] \leftarrow prerequisite[b_j] \cup a_i$;
// A is the set of projects selected by PSP;
$A \leftarrow \text{PSP}(P, profit[], prerequisite[])$;
for $n_i \in N$ **do** // Map PSP solution to node states
  if $a_i \in A$ *and* $b_i \in A$ **then**
    $s[n_i] \leftarrow S_c$;
  else if $a_i \in A$ *and* $b_i \notin A$ **then**
    $s[n_i] \leftarrow S_l$;
  else
    $s[n_i] \leftarrow S_p$;
return $s[]$; // State assignments for nodes in G_W.

Opt-Exec-Plan and PSP.

Integer Linear Programming Formulation. Problem 1 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:

$$X_{a_i} = \mathbb{I}\{s(n_i) \neq S_p\}$$
$$X_{b_i} = \mathbb{I}\{s(n_i) = S_c\}$$

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:

$$\underset{X_{a_i}, X_{b_i}}{\text{minimize}} \quad \sum_{i=1}^{|N|} X_{a_i} l_i + X_{b_i}(c_i - l_i) \tag{3.3a}$$

$$\text{subject to} \quad X_{a_i} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \tag{3.3b}$$

$$\sum_{n_j \in \text{Pa}(n_i)} X_{a_j} - X_{b_i} \geq 0, \quad 1 \leq i \leq |N|, \tag{3.3c}$$

$$X_{a_i}, X_{b_i} \in \{0, 1\}, \; 1 \leq i \leq |N| \tag{3.3d}$$

Equation (3.3b) 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 (3.3c) is equivalent to Constraint 2 — 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 (3.3d) 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 1. Specifically, we set the load cost to ∞ and the compute cost to $-\epsilon$ for a small $\epsilon > 0$. With these values, the cost of a node in S_l, S_p, S_c are $\infty, 0, -\epsilon$ respectively, which makes S_c a clear choice for minimizing Eq(3.3a).

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 3.3d 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 [173]. In fact, it turns out that the above problem is the dual of a flow problem; specifically, it is a minimum cut problem [214, 77]. This motivates the reduction introduced in Section 3.4.2.

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:

$$s(n_i) = \begin{cases} S_c & \text{if } a_i \in A \text{ and } b_i \in A \\ S_l & \text{if } a_i \in A \text{ and } b_i \notin A \\ S_p & \text{if } a_i \notin A \text{ and } b_i \notin A \end{cases} \tag{3.4}$$

Lemma 1. *A feasible solution to PSP under φ 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 φ 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 (3.4), $s(n_i) = S_c$ 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 $a_j \to b_i$. 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 φ. □

Lemma 2. *An optimal solution to PSP is also an optimal solution to OEP under φ.*

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

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} Y_{a_i} p(a_i) + Y_{b_i} p(b_i) \tag{3.5}$$

Substituting our choice for $p(a_i)$ and $p(b_i)$, Eq (3.5) becomes

$$\max_{Y_{a_i}, Y_{b_i}} \sum_{i=1}^{|N|} -Y_{a_i} l_i + Y_{b_i}(l_i - c_i) \tag{3.6}$$

$$= \max_{Y_{a_i}, Y_{b_i}} -\sum_{i=1}^{|N|} (Y_{a_i} - Y_{b_i}) l_i + Y_{b_i} c_i \tag{3.7}$$

The mapping established by Eq (3.4) is equivalent to setting $X_{a_i} = Y_{a_i}$ and $X_{b_i} = Y_{b_i}$. Thus the maximization problem in Eq (3.7) is equivalent to the minimization problem in Eq (3.3a), and we obtain an optimal solution to OEP from the optimal solution to PSP. □

Computational Complexity. For a `Workflow` DAG $G_W = (N_W, E_W)$ in OEP, the reduction above results in $\mathcal{O}(|N_W|)$ projects and $\mathcal{O}(|E_W|)$ prerequisite edges in PSP. PSP has a straightforward linear reduction to Max-Flow [95]. We use the Edmonds-Karp algorithm [56] for Max-Flow, which runs in time $\mathcal{O}(|N_W| \cdot |E_W|^2)$.

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 maybe not be the case due to the intractability of change detection, as discussed in Section 3.3.2. 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 guarantees correctness under the assumption of idempotence, at the cost of some false positives such as $a + b \not\equiv b + a$.

3.4.3 Optimal Materialization Plan

The Opt-Mat-Plan (OMP) problem is tackled by Helix's materialization optimizer: while running workflow W_t at iteration t, intermediate results are selectively materialized for the purpose of accelerating execution in iterations $> t$. We now formally introduce OMP and

show that it is NP-HARD even under strong assumptions. We propose an online heuristic for OMP that runs in linear time and achieves good reuse rate in practice (as we will show in Section 3.5), in addition to minimizing memory footprint by avoiding unnecessary caching of intermediate results.

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

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

$$T_M(W_t) = \sum_{n_i \in M} l_i + T^*(W_{t+1}) \tag{3.8}$$

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 3.4.2, with M materialized.

Equation 3.8 defines the optimization objective for OMP.

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

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

$$\underset{M \subseteq N_t}{\operatorname{argmin}} \sum_{n_i \in M} l_i + T^*(W_{t+1}) \tag{3.9}$$

As discussed in [210], there are many possibilities for W_{t+1}, and they vary by application domain. User modeling and predictive analysis of W_{t+1} itself is a substantial research topic that we will address in future work. This user model can be incorporated into OMP by using the predicted changes to better estimate the likelihood of reuse for each operator. However, even under very restrictive assumptions about W_{t+1}, we can show that OPT-MAT-PLAN is NP-HARD, via a simple reduction from the KNAPSACK problem.

Theorem 3. OPT-MAT-PLAN *is NP-hard.*

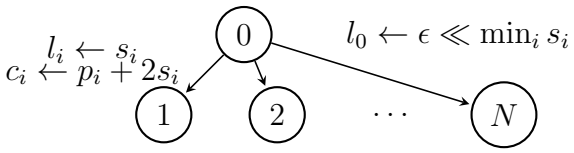


Figure 3.7: OMP DAG for Knapsack reduction.

We show that OMP is NP-hard under restrictive assumptions about the structure of W_{t+1} relative to W_t, which implies the general version of OMP is also NP-hard.

In our proof we make the simplifying assumption that all nodes in the `Workflow` DAG are reusable in the next iteration:

$$n_i^t \equiv n_i^{t+1} \; \forall n_i^t \in N_t, n_i^{t+1} \in N_{t+1} \tag{3.10}$$

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(3.10) 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 $S^* =$*

$$\underset{S \subseteq N}{\operatorname{argmax}} \sum_{i \in S} p_i \tag{3.11}$$

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 3.7. 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 $l_0 = \epsilon < \min s_i$ 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 (3.9) 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., $\sum_{i \in \in \{0,1\}} Y_i l_i \leq B$.

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 (3.9) as

$$\underset{\mathbf{Y} \in \{0,1\}^N}{\operatorname{argmin}} \sum_{i=1}^{N} Y_i l_i + \left(\sum_{i=1}^{N} Y_i l_i + (1 - Y_i) c_i \right) \tag{3.12}$$

Algorithm 2: Streaming OMP

Data: $G_w = (N, E), \{l_i\}, \{c_i\}, \{s_i\}$, storage budget S

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

where $\mathbf{Y} = (Y_1, Y_2, \ldots, Y_N)$. Substituting in our choices of l_i and c_i in terms of p_i and s_i in (3.12), we obtain $\text{argmin}_{\mathbf{Y} \in \{0,1\}^N} \sum_{i=1}^{N} -Y_i p_i$. 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. □

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 3.4.2 is not viable here because of the cold-start problem—materialization decisions need to be made for new operators based on realistic statistics. Thus, to avoid slowing down execution with high memory usage, we impose the following constraint.

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

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

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

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

$$C(n_i) = t(n_i) + \sum_{n_j \in ancestors(n_i)} t(n_j) \tag{3.13}$$

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

Algorithm 2 shows the heuristics employed by Helix's materialization optimizer to decide what intermediate results to materialize. In essence, Algorithm 2 decides to materialize if twice the load cost is less than the cumulative run time for a node. The intuition behind

this algorithm is that assuming loading a node allows all of its ancestors to be pruned, the materialization time in iteration t and the load time in iteration $t+1$ combined should be less than the total pruned compute time, for the materialization to be cost effective.

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 3.6 previously described in Section 3.4.2, are ignored by Algorithm 2—we cannot retroactively update our decision for n_5 after n_8 has been run. We show in Section 3.5 that this simple algorithm is effective in multiple application domains.

Limitations of Streaming OMP. While Streaming OMP performs well on real-world workloads as we will demonstrate in Section 3.5, it can behave poorly in pathological cases. For one simple example, consider a workflow given by a chain DAG of m nodes, where node n_i (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 $\mathcal{O}(m^2)$, whereas a smarter approach would only materialize later nodes and perhaps have storage cost $\mathcal{O}(m)$. If storage is exhausted because Algorithm 2 persists too much early on, this could easily lead to poor execution times in later iterations. We did not observe this sort of pathological behavior in our experiments.

3.4.4 Workflow DAG Pruning

In addition to optimizations involving intermediate result reuse, HELIX further reduces overall workflow execution time by time by pruning extraneous operators from the `Workflow` DAG.

HELIX performs pruning by applying program slicing on the `Workflow` DAG. In a nutshell, HELIX traverses the DAG backwards from the output nodes and prunes away any nodes not visited in this traversal. Users can explicitly guide this process in the programming interface through the `has_extractors` and `uses` keywords, described in Table 3.2. An example of an Extractor pruned in this fashion is `raceExt`(grayed out) in Figure 3.3b), as it is excluded from the `rows has_extractors` statement. This allows users to conveniently perform manual feature selection using domain knowledge.

HELIX provides two additional mechanisms for pruning operators other than using the lack of output dependency, described next.

Data-Driven Pruning. Furthermore, HELIX inspects relevant data to automatically identify operators to prune. The key challenge in *data-driven pruning* is data lineage tracking

across the entire workflow. For many existing systems, it is difficult to trace features in the learned model back to the operators that produced them. To overcome this limitation, Helix performs additional provenance bookkeeping to track the operators that led to each feature in the model when converting DPR output to ML-compatible formats. An example of data-driven workflow optimization enabled by this bookkeeping is pruning features by model weights. Operators resulting in features with zero weights can be pruned without changing the prediction outcome, thus lowering the overall run time without compromising model performance.

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 3.2, 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.

3.5 Empirical Evaluation

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

3.5.1 Systems and Baselines for Comparison

We compare the optimized version of Helix, Helix Opt, against two state-of-the-art ML workflow systems: KeystoneML [177], and DeepDive [221]. In addition, we compare Helix Opt with two simpler versions, Helix AM and Helix NM. While we compare against DeepDive, and KeystoneML to verify 1) and 2) above, Helix AM and Helix NM are used to verify 3). We describe each of these variants below:

	Census (Source: [48])	**Genomics** (Source: [164])	**IE** (Source: [45])	**MNIST** (Source: [177])
Num. Data Source	Single	Multiple	Multiple	Single
Input to Example Mapping	One-to-One	One-to-Many	One-to-Many	One-to-One
Feature Granularity	Fine Grained	N/A	Fine Grained	Coarse Grained
Learning Task Type	Supervised; Classification	Unsupervised	Structured Prediction	Supervised; Classification
Application Domain	Social Sciences	Natural Sciences	NLP	Computer Vision
Supported by Helix	✓	✓	✓	✓
Supported by KeystoneML	✓	✓		✓*
Supported by DeepDive	✓*		✓*	

Table 3.3: 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.

KeystoneML. KeystoneML [177] is a system, written in Scala and built on top of Spark, for the construction of large scale, end-to-end, ML pipelines. KeystoneML specializes in classification tasks on structured input data. No intermediate results are materialized in KeystoneML, as it does not optimize execution across iterations.
DeepDive. DeepDive [221, 45] is a system, written using Bash scripts and Scala for the main engine, with a database backend, for the construction of end-to-end information extraction pipelines. Additionally, DeepDive provides limited support for classification tasks. All intermediate results are materialized in DeepDive.
Helix Opt. A version of HELIX that uses Algorithm 1 for the optimal reuse strategy and Algorithm 2 to decide what to materialize.
Helix AM. A version of HELIX that uses the same reuse strategy as HELIX OPT and *always materializes* all intermediate results.
Helix NM. A version of HELIX that uses the same reuse strategy as HELIX OPT and *never materializes* any intermediate results.

3.5.2 Workflows

We conduct our experiments using four real-world ML workflows spanning a range of application domains. Table 3.3 summarizes the characteristics of the four workflows, described next. We are interested in four properties when characterizing each workflow:

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

Census Workflow. This workflow corresponds to a classification task with simple features from structured inputs from the DeepDive Github repository [48]. It uses the Census Income dataset [50], 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 3.3a). 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 [131]; 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 preprocessing with no specific features but involves multiple learning steps. Both learning steps are unsupervised, which leads to more qualitative and exploratory evaluations of the model outputs 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 [45]. 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 information extraction, and tests a system's ability to handle joins and complex data preprocessing.

MNIST Workflow. The MNIST dataset [108] contains images of handwritten digits to be classified, which is a well-studied task in the computer vision community, from the KeystoneML [177] evaluation. The workflow involves nondeterministic (and hence not reusable) data preprocessing, with a substantial fraction of the overall run time spent on L/I in a typical iteration. We include this application to ensure that in the extreme case where there is little reuse across iterations, HELIX does not incur a large overhead.

Each workflow was implemented in HELIX, and if supported, in DeepDive and KeystoneML, with ✓* in Table 3.3 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`
- IE DeepDive: `https://github.com/HazyResearch/deepdive/blob/master/examples/spouse/app.ddlog`
- MNIST KeystoneML: `https://github.com/amplab/keystone/blob/master/src/main/scala/keystoneml/pipelines/images/mnist/MnistRandomFFT.scala`

DeepDive has its own DSL, while KeystoneML's programming interface is an embedded DSL in Scala, similar to HML. We explain limitations that prevent DeepDive and KeystoneML from supporting certain workflows (grey cells) in Section 3.5.5.1.

3.5.3 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 [210] 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 in Section 2.1.3.

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.

3.5.4 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.

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

3.5.5.1 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 3.3 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, fine-grained features are cumbersome to program since operators are designed to operate on the entire dataset, making it not suitable for structured prediction tasks due to complex data preprocessing. On the other hand, DeepDive is highly specialized for information extraction and focuses on supporting data preprocessing. Unfortunately, its learning and evaluation components are not configurable by the user, limiting the type of ML tasks supported. DeepDive is therefore unable to support the MNIST and genomics workflows, both of which required custom ML models. Additionally, we are only able to show DeepDive performance for DPR iterations for the supported workflows in our experiments.

3.5.5.2 Cumulative Run Time

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

Figure 3.8 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 3.9 shows the breakdown by workflow components and materialization for the individual iteration run times in HELIX, with the same color scheme as in Figure 3.8 for the workflow components and gray for materialization time.

Census. As shown in Figure 3.8(a), the census workflow has the largest cumulative run time gap between HELIX OPT and the competitor systems—HELIX OPT *is* **19**× *faster than KeystoneML as measured by cumulative run time over 10 iterations.* By materializing and reusing intermediate results HELIX OPT is able to substantially reduce cumulative run-time relative to other systems. Figure 3.9(a) shows that 1) on PPR iterations HELIX recomputes only the PPR; 2) the materialization of L/I outputs, which allows the pruning of DPR and L/I in PPR iterations, takes considerably less time than the compute time for DPR and L/I; 3) HELIX OPT reruns DPR in iteration 5 (L/I) because HELIX OPT avoided materializing the large DPR output in a previous iteration. For the first three iterations, which are DPR (the only type of iterations DeepDive supports), the 2× reduction between HELIX OPT and

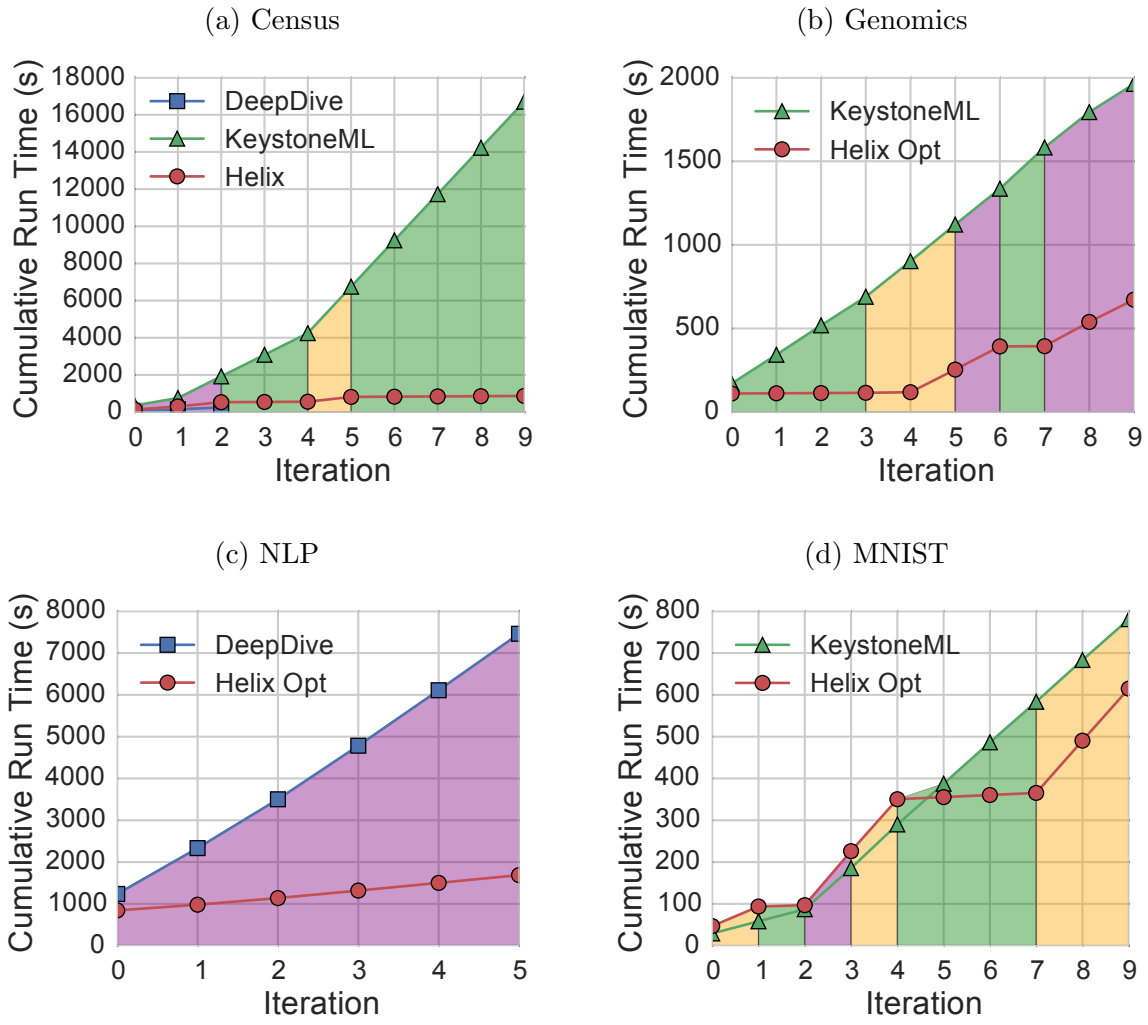


Figure 3.8: Cumulative run time for the four workflows. The color under the curve indicates the type of change in each iteration: purple for DPR, orange for L/I, and green for PPR.

DeepDive is due to the fact that DeepDive does data preprocessing with Python and shell scripts, while HELIX OPT uses Spark. While both KeystoneML and HELIX OPT use Spark, KeystoneML takes longer on DPR and L/I iterations thanHELIX OPT due to a longer L/I time incurred by its caching optimizer's failing to cache the training data for learning. 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 [210].

Genomics. In Figure 3.8(b), HELIX OPT shows a **3**× speedup over KeystoneML for the genomics workflow. The materialize-nothing strategy in KeystoneML clearly leads to no run time reduction in subsequent iterations. HELIX OPT, on the other hand, shows a per-iteration run time that is proportional to the number of operators affected by the change in

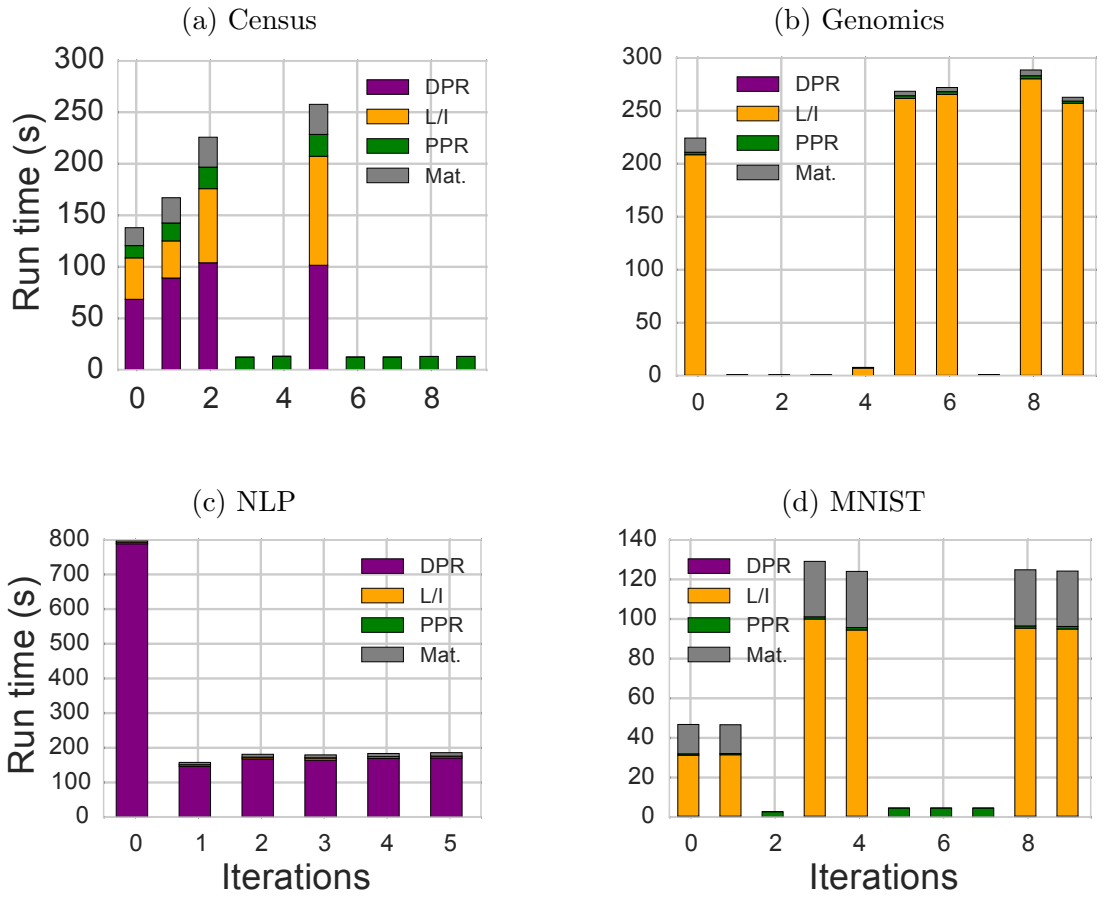


Figure 3.9: Run time breakdown by workflow component and materialization time per iteration for Helix.

that iteration. Figure 3.9(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 3.8(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 3.9(c), leading to large run time reductions in iterations 1-5 and thus a

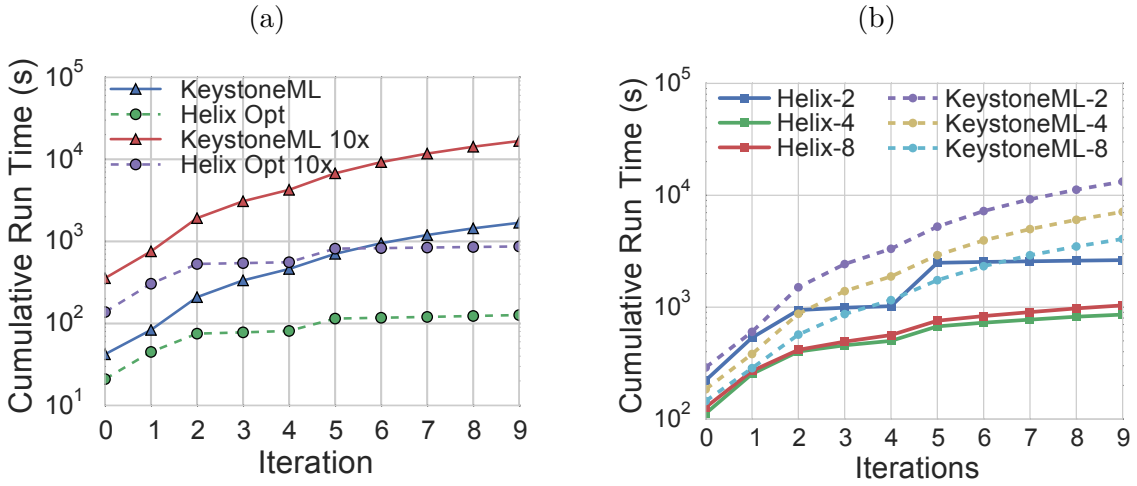


Figure 3.10: a) Census and Census 10x cumulative run time for HELIX and KeystoneML on a single node; b) Census 10x cumulative run time for HELIX and KeystoneML on different size clusters.

large cumulative run time reduction.

MNIST. Figure 3.8(d) shows the cumulative run times for the MNIST workflow. As mentioned above, the MNIST workflow has nondeterministic data preprocessing, which means any changes to the DPR and L/I components prevents safe reuse of any intermediate result. However, iterations containing only PPR changes can reuse intermediates for DPR and L/I had they been materialized previously. Furthermore, we found that the DPR run time is short but cumulative size of all DPR intermediates is large. Thus, materializing all these DPR intermediates would incur a large run time overhead. KeystoneML, which does not materialize any intermediate results, shows a linear increase in cumulative run time due to no reuse. HELIX OPT, on the other hand, only shows slight increase in runtime over KeystoneML for DPR and L/I iterations because it is only materializing the L/I results on these iterations, not the nonreusable, large DPR intermediates. In Figure 3.9(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.

3.5.5.3 Scalability vs. KeystoneML

Dataset Size. We test scalability of HELIX and KeystoneML with respect to dataset size by running the ten iterations in Figure 3.8(a) of the Census Workflow on two different sizes of the input. Census 10x is obtained by replicating Census ten times in order to preserve the learning objective. Figure 3.10(a) shows run time performance of HELIX and KeystoneML

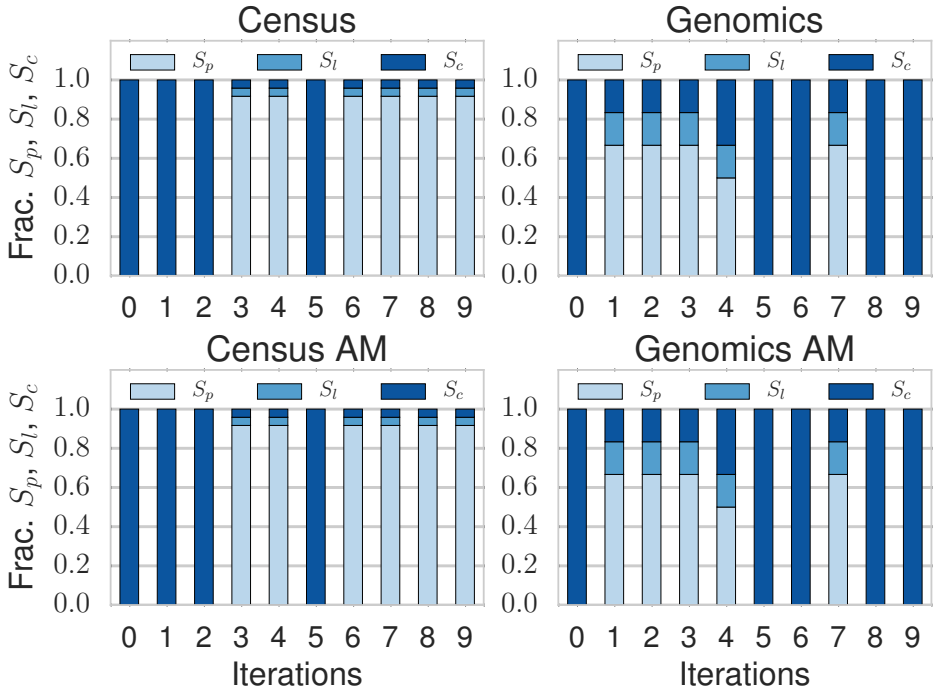


Figure 3.11: Fraction of states in S_p, S_l, S_c as determined by Algorithm 1 for the Census and Genomics workflows for HELIX OPT and HELIX AM.

on the two datasets on a single node. 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 3.8(a) on Census 10x described above. Using a uniform set of machines, we create clusters with 2, 4, and 8 workers and run HELIX and KeystoneML on each of these clusters to collect cumulative run time.

Figure 3.10(b) shows that 1) HELIX has lower cumulative run time than KeystoneML on the same cluster size, consistent with the single node results; 2) KeystoneML achieves $\approx 45\%$ run time reduction when the number of workers is doubled, scaling roughly linearly with the number of workers; 3) From 2 to 4 workers, HELIX achieves up to 75% run time reduction 4) From 4 to 8 workers, HELIX sees a slight increase in run time. Recall from Section 3.2 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

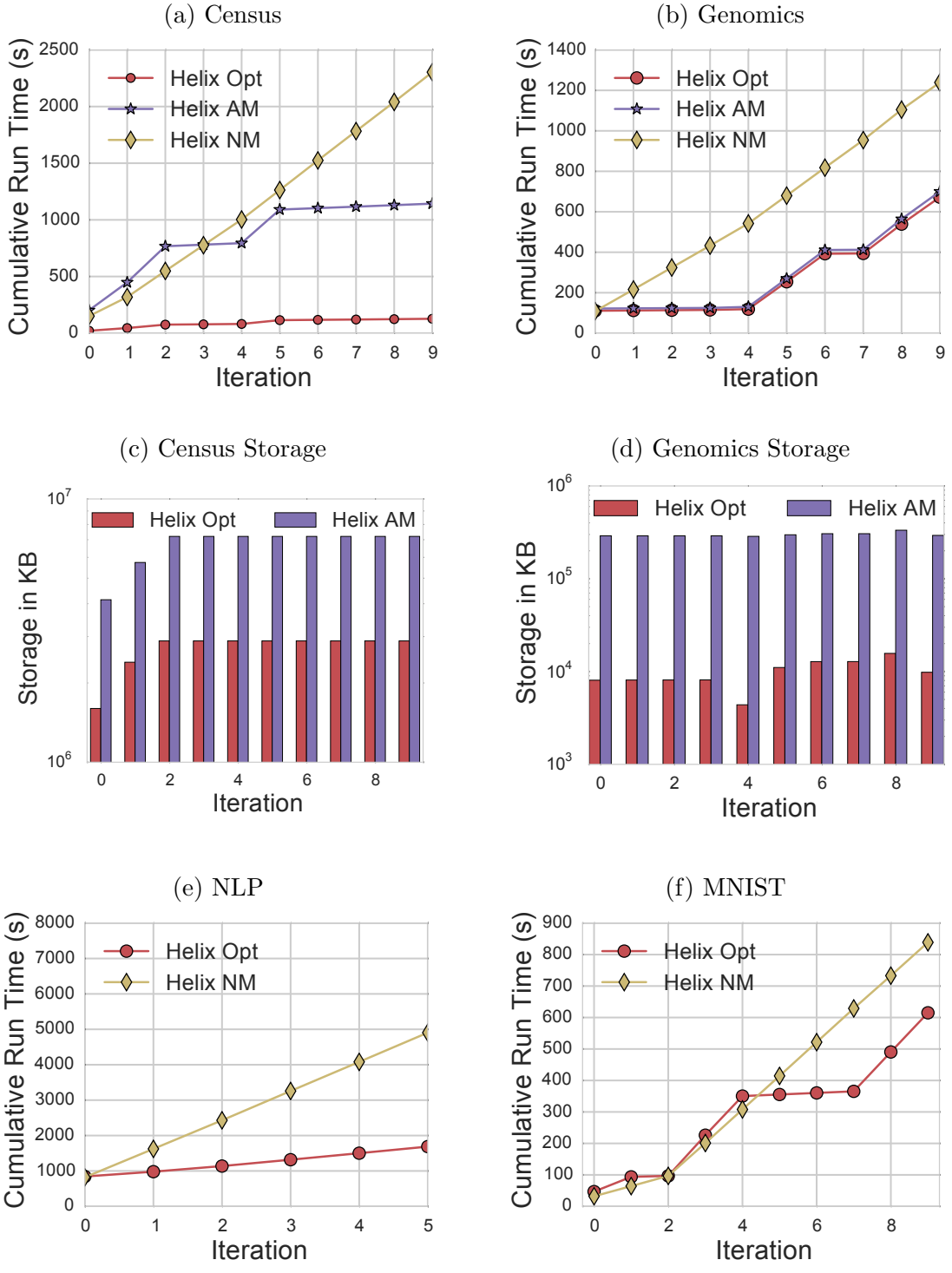


Figure 3.12: Cumulative run time and storage use against materialization heuristics on the same four workflows as in Figure 3.8.

operations outweighs the benefits of distributed computing, hence the slight increase in 4).

3.5.6 Evaluation vs. Simpler Helix Versions

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

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

Figures 3.12(a), (b), (e), and (f) show the cumulative run time on the same four workflows as in Figure 3.8 for the three variants.

HELIX AM *is absent from Figures 3.12(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 3.12(e) and (f) show that HELIX OPT achieves substantial run time reduction over HELIX NM using very little materialization time overhead (where the red line is above the yellow line).

For the census and genomics workflows where the materialization time is not prohibitive, Figures 3.12(a) and (b) show that in terms of cumulative run time, HELIX OPT outperforms HELIX AM, which attains the best reuse as explained above. We also compare the storage usage by HELIX AM and HELIX NM for these two workflows. Figures 3.12(c) and (d) show the storage size snapshot at the end of each iteration. The x-axis is the iteration numbers, and the y-axis is the amount of storage (in KB) in log scale. The storage use for HELIX NM is omitted from these plots because it is always zero.

We find that HELIX OPT outperforms HELIX AM while using less than half the storage used by HELIX AM for the census workflow in Figure 3.12(c) and $\frac{1}{30}$ the storage of HELIX AM for the genomics workflow in Figure 3.12(d). Storage is not monotonic because HELIX purges any previous materialization of original operators prior to execution, and these operators may not be chosen for materialization after execution, thus resulting in a decrease in storage.

Furthermore, to study the optimality of Algorithm 2, we compare the distribution of nodes in the prune, reload, and compute states S_p, S_l, S_c between HELIX OPT and HELIX AM for workflows with HELIX AM completed in reasonable times. Since everything is materialized in HELIX AM, it achieves maximum reuse in the next iteration. Figure 3.11

shows that HELIX OPT enables the exact same reuse as HELIX AM, demonstrating its effectiveness on real workflows.

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

3.6 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* [27]. Systems that support *declarative ML algorithms*, such as TensorFlow [1], SystemML [65], OptiML [181], ScalOps [200], and SciDB [179], allow ML experts to program new ML algorithms, by declaratively specifying linear algebra and statistical operations at higher levels of abstraction. Although it also builds a computation graph like HELIX, TensorFlow has no intermediate reuse and always performs a full computation e.g. any in-graph data preparation. TensorFlow's lower level linear algebra operations are not conducive to data preprocessing. HELIX handles reuse at a higher level than TensorFlow ops. *ML libraries*, such as Mahout [145], Weka [73], GraphLab [115], Vowpal Wabbit [106], MLlib [126] and Scikit-learn [151], provide simple interfaces to optimized implementations of popular ML algorithms. TensorFlow has also recently started providing TFLearn [42], a high level ML library targeted at deep learning. Systems that support *declarative ML tasks* allow application developers with limited ML knowledge to develop models using higher-level primitives than in declarative ML algorithms. HELIX falls into this last group of systems, along with DeepDive [221, 45] and KeystoneML [177]. 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 [75, 59, 196] that extend databases to support ML.

Declarative ML task systems, like HELIX, can seamlessly make use of improvements in ML library implementations, such as MLlib [126], CoreNLP [120] and DeepLearning4j [49], within UDF calls. Unlike declarative ML algorithm systems, that are targeted at ML experts and researchers, these systems focus on end-users of ML.

Systems that Optimize Across Frameworks. These systems target a broad range of use-cases, including ML. Weld [147] and Tupleware [41] optimize UDFs written in different frameworks by compiling them down to a common intermediate representation. Declarative

ML task systems like HELIX can take advantage of the optimized UDF implementations; unlike HELIX, these systems do not benefit from seamless specification, execution, and end-to-end optimizations across workflow components that come from a unified programming model.

Systems for Optimizing Data Preprocessing. The database community has identified various opportunities for optimizing DPR. Several approaches identify as a key bottleneck in DPR and optimize it [103, 36, 143, 105]. Kumar et al. [103] optimizes generalized linear models directly over factorized / normalized representations of relational data, avoiding key-foreign key joins. Morpheus [36] and F [143] extend this factorized approach to general linear algebra operations and linear regression models, respectively (the latter over arbitrary joins). Some work [105] 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 in data cleaning and feature engineering [162, 222, 99, 14, 13]. All of these optimizations are orthogonal to those used by HELIX, which targets end-to-end iterative optimizations. Snorkel [162] supports training data engineering using rules. COLUMBUS [222] optimizes feature selection specifically for regression models. ActiveClean [99] integrates data cleaning with learning convex models, using gradient-biased samples to identify dirty data. Brainwash [14] proposes to expedite feature engineering by recommending feature transformations. Zombie [13] speeds up data preparation by learning over smaller, actively-learned informative subsets of data during feature engineering. These approaches are bespoke for the data preprocessing 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 [149] and Mistique [194] 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 [22, 54, 19, 124, 113, 217], 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. However, as we will discuss in Chap-

ter 4, TFX has begun to gain industry-wide traction and is now capable of supporting more general-purpose workloads.

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 workflow HELIX manages sits at a lower level than those of industry PaaS systems, and therefore the techniques it leverages are quite different.

General Batch Processing Workflow Systems. A number of companies have implemented workflow management systems for batch processing [23, 182]. 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 [33] and Kepler [116] 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 [215]. One such system, Pegasus [47], also identifies reuse opportunities when executing workflows. The optimization techniques employed by all systems discussed leverage reuse in a simpler manner than does 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 [38], but operate 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. COLUMBUS [222] focuses on caching feature columns for feature selection exploration within a single workflow. ReStore [57] manages reuse of intermediates across dataflow programs written in Pig [141], while Nectar [70] does so across DryadLINQ [216] workflows. Jindal et al. [84] study SQL subexpression materialization within a single workflow with many subqueries. Perez et al. [152] 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 [194] and its spiritual predecessor Sherlock [195] use historical usage as part of their cost models for adaptive materialization. HELIX shares some similarities with the systems above but also differs in significant ways. Mistique [194], Nectar [70], and ReStore [57] 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 do not consider the operator/subquery dependencies in the DAG and make decisions for each operator independently based on availability, operator type, size, and compute time. We have shown in Figure 3.6 that decisions can have cascading effects on the rest of the

workflow. The reuse problems studied in COLUMBUS [222] and Jindal et al. [84] 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 [194] and Perez et al. [152] 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.

3.7 Conclusion

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 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.

Post model development, the next stage in the ML lifecycle is model deployment, which has a different set of requirements and challenges from the development phase. We delve into in deployment in Chapter 4.

Chapter 4

Understanding Production Pipelines

As introduced in Section 1.2 in Chapter 1, the model deployment, also commonly referred to as model productionization, stage comes after model development. Many industry ML practitioners have written, usually anecdotally, about the infrastructure and practices at their organizations for productionizing ML models [175, 21, 35]. While each organization has a slightly different setup, the consensus across the board is that ML production pipelines are complex, with many interlocking analytical components beyond training. This has spurred on the development of many end-to-end ML systems (e.g., TFX [21], MLFlow [218], Microsoft Azure ML [201], AWS Sagemaker [113]) and open-source ML libraries (e.g., MLlib [117], MetaFlow [127], and Scikit-Learn [176]), all of which provide native support for data preprocessing, data validation, model validation, and model deployment, in addition to modeling training, all within a single environment.

As an example, TFX [21] includes pipeline steps that perform different flavors of data analysis and transformation, or of data- and model-validation, both before and after the training step. The topology of the corresponding graph can be complicated. For instance, model chaining (where a model is used to generate data for another model) is becoming increasingly common, introducing model-to-model dependencies in the same pipeline. And of course, there is the step of deploying a model after training, in a scalable serving infrastructure. Moreover, production ML pipelines often work in a continuous mode, with periodic retraining and deployment as fresh data becomes available. Overall, the steps across these pipelines interact in complex ways, and their compound effects might be hard to predict or debug, necessitating the management of provenance across them. Provenance management is one of the key value adds of existing end-to-end ML platforms, such as TFX [22], MLFlow [218] or MetaFlow [127].

While there is anecdotal evidence for these end-to-end concerns beyond training, little is known about production ML deployments and the challenges they surface in terms of data management:

- **Coarse-grained pipeline characteristics.** What do production ML pipelines look like in a large data-driven organization? What types of models are used? What types of

feature engineering procedures are used, and how complex are they from a data processing standpoint? What is the lifespan of typical pipelines?

- **Fine-grained pipeline characteristics.** How much overlap exist between executions of a given pipeline? For portions of a pipeline that are executed repeatedly to derive models, how does the data distribution change? How often are they executed, and how often are the resulting models deployed?
- **Opportunities.** Are there uniform ways to represent and reason about these pipelines? Any opportunities to make these pipelines more efficient, e.g., by leveraging sharing of computation, pruning redundant computation, making more efficient use of system resources, and leveraging incremental view maintenance?

Answering these questions can improve our understanding of production ML. In turn, this increased awareness can help the research community move beyond training efficiency to more effectively supporting the end-to-end production ML pipelines. This involves addressing new incarnations of familiar database research challenges—from efficient data preparation and cleaning, to optimized query plans, to dealing with streaming data, to sharing of computation, to materialization and reuse, to provenance for reproducibility and debugging.

In this chapter, we take a first step in this direction by *analyzing a large corpus of 3000 production ML pipelines at Google, comprising over 450,000 trained models, over a period of four months.* To the best of our knowledge, no similar corpus has ever been analyzed in prior literature. This unique corpus of thousands of TensorFlow Extended (TFX) pipelines, with hundreds of thousands of generated models, spans different modalities (tabular data, video and text embeddings, personalization), tasks (regression, classification), and environments (production, development). Our analysis reveals a number of interesting insights, including the fact that: *(a)* training accounts for only 20% of the total computation time, despite $\sim$ 60% of models being deep neural nets (DNNs); *(b)* the rest 40% of the pipelines train traditional, non-DNN, ML models, showing the value of simpler model architectures in production, but also the need to manage a diverse set of model types; *(c)* the input data used for consecutive model updates have large overlaps but also significant differences in data distribution, underlining the need to cope with data and concept drift, *(d)* models in each pipeline are updated 7 times per day on average, with a substantial fraction (1.12%) of pipelines updating models over 100 times a day(!), giving rise to potential instability that require special care, *(e)* only one in four model retraining results in model deployment, with the three undeployed model updates representing wasted computation, discussed more later. While our results stem from analyzing production ML pipelines at Google, due to the commonalities between TFX and other end-to-end ML platforms, as well as the adoption of TFX in other large organizations [91], we expect our findings to generalize to other production use cases and thus be of general interest. Additionally, we hope our approach to analyzing this complex corpus can serve as inspiration for studying other corpora of ML systems history.

Specifically, we make the following contributions:

Coarse-grained analysis of pipeline lifespan, components, architecture, and com-

plexity (Section 4.3). We provide the first-ever study of 3000 ML production pipelines captured over a four-month period to understand the underlying data management challenges. This analysis surfaces coarse-grained characteristics about these pipelines, such as their lifespan in the end-to-end training of several models, proportion of resources devoted to data analytics (beyond training), and the features, feature transformations, and model architectures in the pipelines.

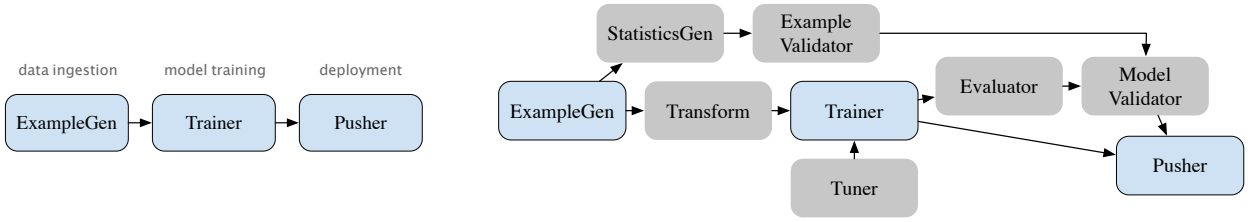
Model graphlet abstraction and fine-grained analysis of frequency, failure, and overlap (Section 4.4). We proceed to analyze finer-grained properties of these pipelines through provenance analysis. Many previous studies analyzed provenance graphs in data workflows, but the complexity and unique characteristics of production ML necessitate a new approach. We introduce the notion of *model graphlets*, wherein the provenance graph is decomposed into sub-graphs to capture the end-to-end execution of the pipeline including several instances of model training. One can view model graphlets as a ML-oriented application of the more general concept of provenance segmentation [3]. We then characterize these graphlets in terms of their lifespan, complexity, overlap, failure points, and their connection to model deployment.

4.1 Related Work

We briefly cover previous studies related to tracking, analyzing, and optimizing ML pipelines in production.

Provenance management and analysis. Provenance for complex systems has been studied for relational databases [37], scientific workflows systems [62], dataflow systems [12, 82], data lakes [72, 74], and ML systems [218, 193, 128]. Previous work has even led to the standardization of provenance representations for workflows in the form of graphs [80, 136, 135]. Other research has proposed various ways to explore and analyze such provenance graphs, e.g., visualization [20], reachability query support [18], support for user-defined views [26], segmentation and summarization [3, 7, 129]. Our work introduces a framework to segment ML provenance graphs and demonstrates how this segmentation leads to further analysis and optimizations for ML pipelines.

Metadata tracking for the ML Lifecycle. In modern end-to-end ML systems, e.g., TFX [21], MLFlow [218], Kubeflow [100], and MetaFlow [127], data science development tools, e.g., ModelDB [193], noWorkflow [153], and Pachyderm [146], as well as model development practices in the industry, e.g., [171, 169], there is an effort towards tracking metadata to facilitate workflow orchestration and aid model reproducibility over the ML project lifecycle. These metadata tracking solutions vary in terms of ingestion method (user input vs. transparent), data model (relational vs. graph), and scope of the metadata (training vs. end-to-end). Our work here is based on a specific framework, MLMD [132] in TFX [22, 186], which automatically records the end-to-end provenance using a graph-based model. We build on this framework and introduce a method to segment large provenance graphs into smaller model-centric graphs. However, our analysis and this decomposition is orthogonal



(a) A simple TFX pipeline comprising three operators: *ExampleGen*, *Trainer*, and *Pusher*. Edges denote the input/output dependencies.

(b) A more typical TFX pipeline that comprises additional operators for data preprocessing, data and model validation, and tuning. Shaded operators correspond to the additional functionality compared to the simple pipeline in (a).

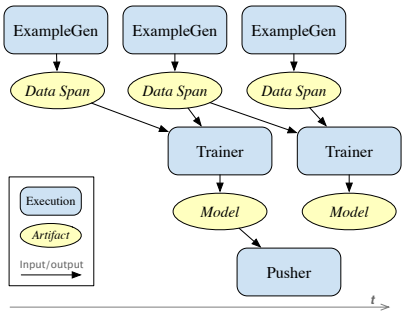
Figure 4.1: Examples of TFX pipelines

from the specific representation of complete provenance, and the same methodology could apply to other systems. Moreover, our final contribution with respect to pruning redundant graphlets has not been explored in previous work to the best of our knowledge.

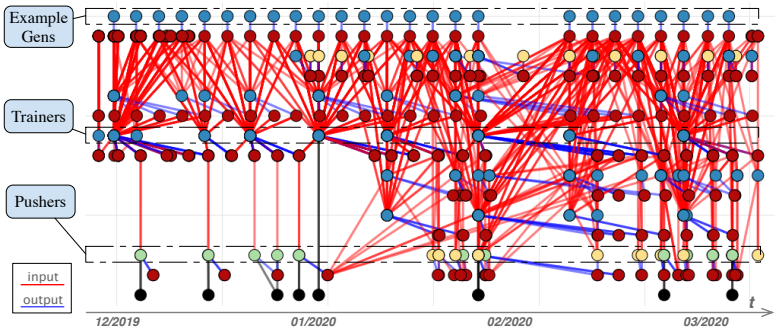
Understanding ML workflows. While our work is, to the best of our knowledge, the first large-scale study of production ML pipelines, a few prior works have conducted empirical studies on ML and data science (DS) workflows. The work presented in Chapter 2 aims to understand how ML developers iterate on models by studying ML workflows generated by novice and intermediate ML developers on Kaggle-style tasks with static datasets and no model deployment. Our study in this chapter is fundamentally different in the nature of the ML tasks studied—with an emphasis on production ML and pipelines that are run over a long period. Some works in the HCI community study ML/DS workflows by interviewing ML developers and data scientists [220, 88, 11]. Another related area of work is anecdotal reports and retrospectives by industry ML practitioners [155, 172, 175] that shed light on real-world ML practices and challenges. We complement this line of work with quantitative insights from a corpus of production ML pipelines.

4.2 Preliminaries

Our corpus comprises TensorFlow Extended (TFX) [22] pipelines. TFX is an end-to-end platform for production ML used by product teams across Google. The platform has been recently open-sourced and has been adopted by major organizations outside Google as well [91]. TFX provides a set of operators that can be strung together into a pipeline that conceptually accepts data as input and produces a model as output. (The actual topology of the pipeline can get much more complicated in practice, as we describe later.) Even though our discussion here is centered around TFX (to match the actual corpus), we note that the concepts that we introduce are present in other end-to-end ML frameworks discussed previously.



(a) A simple trace for the pipeline in Fig. 4.1(a).



(b) A real-world trace from our corpus, using operators shown in Fig. 4.1(b).

Figure 4.2: Examples of pipeline traces. The left-to-right order preserves the time of artifact generation.

4.2.1 Basic Concepts

We use the term ***pipeline*** to represent a graph of operators that are connected in a producer/consumer fashion. Figure 4.1(a) shows a simple TFX pipeline comprising three operators: *ExampleGen*, which imports data in a format suitable for training; *Trainer*, which uses the imported data to train a model; and *Pusher*, which takes the generated model and deploys it for inference in some external service (e.g., TensorFlow.Serving[1]). The topology of the pipeline graph corresponds to the input/output relationships between operators. Moreover, we assume that these relationships are "type-checked", e.g., *ExampleGen* outputs data in a format that is understood by the *Trainer*, when the pipeline is authored. The specifics of pipeline authoring are orthogonal to our work.

In itself, a pipeline encodes only coarse-grained dependencies between its operators. In contrast, a pipeline ***trace*** records the fine-grained relationship between individual executions of the operators and the provenance of their input/output artifacts when the pipeline runs in production. Formally, a trace is a directed acyclic graph of *execution* nodes and *artifact* nodes, with edges linking execution nodes to their input/output artifacts. Figure 4.2(a) shows a sample trace of the pipeline in Figure 4.1(a), where the nodes are arranged horizontally, left to right, in increasing order of the finish time for executions and creation time for artifacts. In this trace, the *ExampleGen* operator has executed three times, producing one *data span* artifact per execution. A *data span* artifact corresponds to a chunk of data whose semantics depend on the pipeline. For instance, if the pipeline trains models to recommend videos to users, then a single data span might correspond to previous user interactions with the video service over the period of a single day. Continuing with the example, the *Trainer* operator has executed two times, each time reading the last two data spans in the pipeline and producing the corresponding model. We note that this pattern is quite common in

[1] https://github.com/tensorflow/serving

practice: there is a data ingestion process (here, the *ExampleGen* operator) that produces outputs at a fine level of granularity (e.g., each span corresponds to a single day of data) and the model training process reassembles the data into coarser granularity (e.g., a rolling window of the last two days). The example trace concludes with the *Pusher* operator, which executes once for the first model but not for the second model, which means that the second model was not deployed. The latter case is not uncommon in production and can be attributed to several reasons, e.g., the model does not have better performance than the previous model or fails certain validation checks, or the deployment mechanism throttles model pushes to avoid overloading.

Note that a trace does not reflect any information about the orchestration of the pipeline operators. For instance, the example trace might be generated through a serial execution of operators, or by asynchronous scheduling where several operators with overlapping inputs can run at the same time. Moreover, a pipeline may be triggered periodically (e.g., by ingesting the newest span of data every hour and triggering new runs of the operators) or manually (e.g., a model developer reruns the pipeline after making changes to the input data or training code). All we assume is that some external system is responsible for scheduling operators, and the trace will grow over time with every run of the pipeline to contain the full history of operator executions and generated artifacts.

Up to this point, we used a simple example comprising the most elemental steps of an ML production pipeline: data ingestion, training, and deployment. In practice, pipelines are more complicated. First, there are several other operators that correspond to important stages and safety checks in production ML. Figure 4.1(b) shows a more typical pipeline that expands on the simple pipeline of Figure 4.1(a) by including operators for data analysis and validation, data pre-processing, and model analysis and validation. Pipeline authors may also introduce custom operators depending on their ML task. Second, these operators can be wired in different topologies, e.g., a Trainer operator might generate an initial model that is then distilled through a separate Trainer operator to produce the final model, or the data-validation operator might block the execution of downstream operators if the data contains any errors. Last, the configuration of the pipeline may change over time. For instance, the pipeline might start with training a single "production" model, then be augmented over time with more Trainer operators that correspond to "experiment" models (some of which might become production models eventually).

This pipeline-level complexity carries over to the recorded traces. Moreover, it is common to have long-running production pipelines that continuously ingest data spans and output "fresh" models, resulting in traces that continuously grow over time. Add to that the fact that executions often share artifacts (e.g., through the rolling-window mechanism described in Figure 4.2(a)), and it becomes clear that traces can easily become large graphs with complicated structure that are challenging to analyze. Figure 4.2(b) shows one such large real-world trace; our corpus has pipelines with as many as **6900 nodes**. This observation motivates our proposal to analyze traces through lower-complexity or finer-grained *model graphlets* that essentially "unnest" each trace graph with respect to the generated models—more on this in Section 4.4.

4.2.2 Corpus of Traces for Analysis

The main contribution of this paper is the first-ever analysis of a corpus of production ML pipelines. The corpus comprises the traces of 3000 TFX pipelines (each with up to 6900 nodes) deployed at Google over a period of four months. We focused on pipelines that generated at least one trained model and had at least one model deployed outside the pipeline. These are the production pipelines whose models support downstream applications. The resulting pipelines correspond to hundreds of teams that span product areas (e.g., advertising, video recommendations, app recommendations, and maps), ML tasks (e.g., single-/multi-label classification, regression, and ranking), and model architectures (linear and deep models of varying complexity). In total, the collected traces contain 7.7M execution nodes and 20M artifact nodes.

The traces were collected using the ML Metadata [132] framework (MLMD), which powers the management of metadata and provenance in TFX [21]. Similar to our earlier definition, MLMD models a pipeline trace as a graph of *Execution* and *Artifact* nodes with their input/output relationships. We note that the recorded MLMD traces also carry *Context* nodes to represent the grouping of *Artifact*s and *Execution*s, but this information is not used in our analysis. Note that provenance graph in MLMD is different from the operator DAG presented in Section 3.3.1 in Chapter 3, where the nodes are the operators instead of executions and artifacts. While TFX also has a notion of the operator DAG in the pipeline configuration, MLMD tracks executions instead since there is control flow that can skip operators during execution. MLMD records additional metadata per node (e.g., start and completion time of *Execution*s, creation time of *Artifact*s), which we use to glean information about the triggering cadence of pipelines. Moreover, the corpus records high-level information for each data span artifact, comprising the features present in the span, their types, and statistics for different feature types (e.g., the unique values for a categorical feature, or the mean and standard deviation for a numerical feature). It's worth mentioning that both TFX and MLMD are open sourced and similar traces with those operators can be derived by other TFX users.

4.3 Pipeline-level Analysis

We begin with an analysis of the corpus at the level of entire traces, aiming to understand higher-level characteristics of ML pipelines, such as the structural properties of the corresponding graphs, common data types and transformations, common ML model architectures, and the cost of different operators in addition to training. Section 4.4 will present a finer-grained analysis at the level of sub-traces, to understand the behavior of ML pipelines from the perspective of individual models.

4.3.1 Pipeline Lifespan and Activity

We first examine two aspects of ML pipelines: their typical lifespan and model training frequency.We define the lifespan of a pipeline as the count of days between the timestamps of the newest and the oldest nodes in its trace. Hence, the lifespan is an indication of how long the pipeline is active. The distribution in Figure 4.3(a) shows that, on average, pipelines are active for 36 days, with some being active for the entire span of our corpus (130 days). The longer lifespans correspond to continuous pipelines that generate a stream of models based on a stream of incoming data and are common within product groups that are heavy users of ML. Figure 4.3(d) shows the distribution of lifespan broken down by model type, where "DNN" refers to all deep learning models, "Linear" refers to all generalized linear models, and "Rest" contains all other models, including tree-based models. Interestingly, the pipelines with linear models live longer than the ones with DNN models.

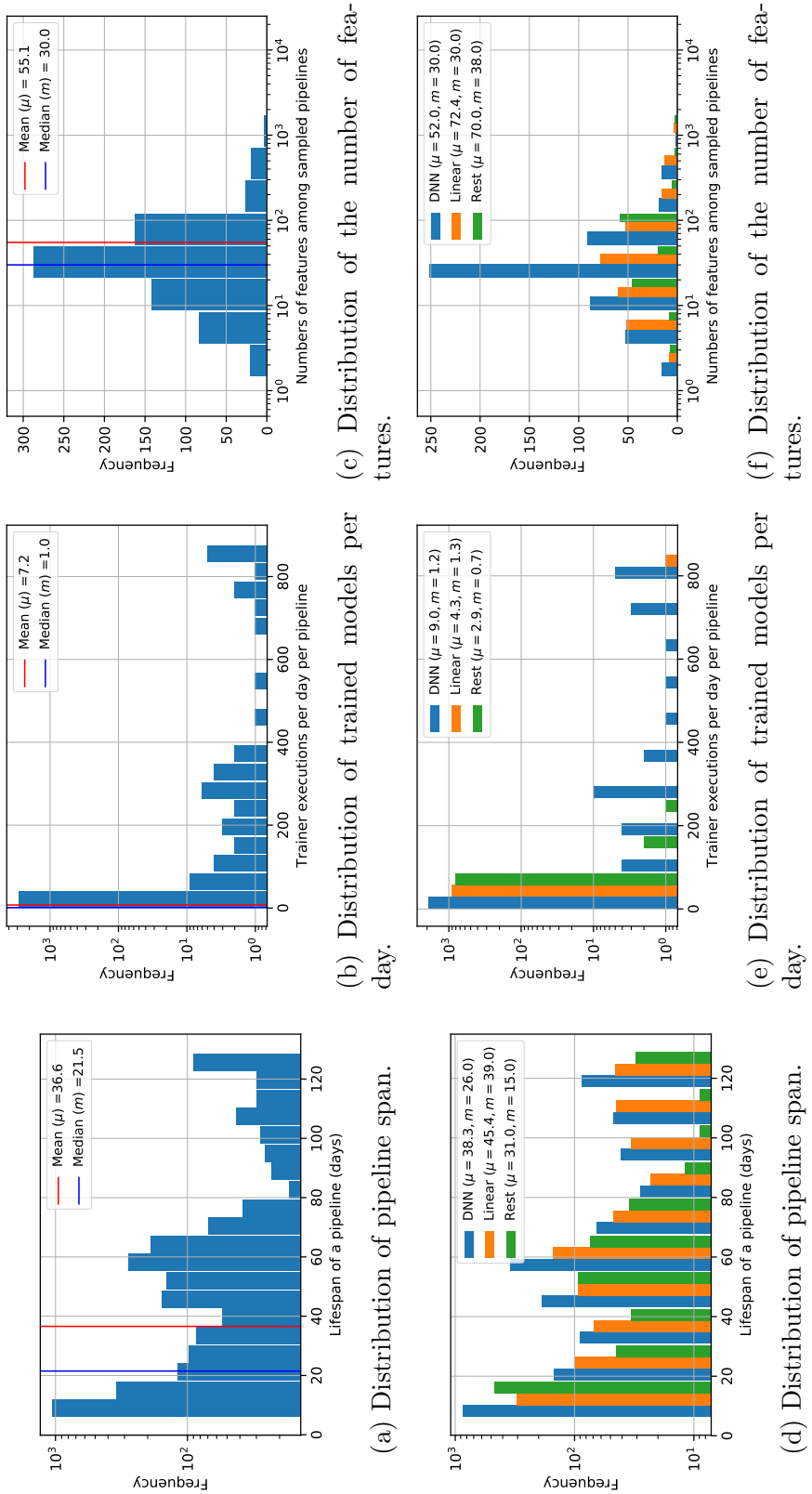
Figure 4.3(b) shows a different dimension of pipeline activity in terms of the average number of trained models in the trace, per day. The majority of pipelines generate one model per day but there is also a wide spread of cadences, with some pipelines training close to a thousand models per day! Upon closer investigation, these are continuous pipelines that have a fast stream of incoming data and the ability to quickly produce updated models. As in the previous figure, these pipelines correlate well with product teams that are heavy users of ML. In addition, in Figure 4.3(e), we find that the cadence of pipelines using DNN are more diverse than the other methods, which reflect a wide adoption of it in different problems.

Overall, pipelines can have a large lifespan and generate models at a high rate. In turn, the accumulated state of these pipelines can become complex. For instance, the trace of the more complicated pipelines can have up to 6953 (artifact or execution) nodes. Tools to efficiently query or summarize these complex traces can become indispensable for humans to debug or manage these pipelines.

4.3.2 Pipeline Complexity

Next, we analyze the complexity of ML pipelines. Specifically, we examine three aspects: 1) the shape of the input data; 2) the typical transformations for training; 3) the diversity of model architectures.

Input Data Shape. Figure 4.3(c) and Figure 4.3(f) show the distributions of the feature count in the input data. We use "feature" to refer to a column in the data (e.g., the 'video-id' column) and "domain" for the set of values that this feature has in the input examples. We compute the distribution of feature count as follows: we identify the data span nodes in all traces, and use the associated MLMD metadata for each data span (Section 4.2.2) to retrieve the number of features. As shown, the vast majority of the pipelines utilize up to 100 features. However, higher feature counts do appear and there are extreme cases of tens of thousands of features in a pipeline.



(a) Distribution of pipeline span.

(b) Distribution of trained models per day.

(c) Distribution of the number of features.

(d) Distribution of pipeline span.

(e) Distribution of trained models per day.

(f) Distribution of the number of features.

Figure 4.3: Pipeline Activity and Data Complexity Analysis Results.

We also examine the composition of features in each pipeline in terms of numerical (e.g., length of a video) and categorical/sparse features (e.g., id of a video, query text). Note that this distinction does not necessarily reflect the type used to encode the feature. For instance, a 'video id' feature might be encoded as a number but treated as a sparse/categorical feature in training as well as in our breakdown. We find that the features of each pipeline are equally distributed among these two types: on average, 53% of the features are categorical. Moreover, we analyzed the domain of the categorical features and found that each categorical feature has, on average, **10.6 million** unique values in its domain. For pipeline with DNN models, the average is 13.6 millions, while for the ones having Linear models, it is larger than 20 million. This large domain size indicates high data complexity and thus increased cost and complexity for data transformations prior to training. For instance, we discuss below how such categorical features are typically embedded in vocabularies prior to training.

Feature Transformation. The additional metadata that we collect in our corpus (see Section 4.2.2) provides a glimpse into the types of transformations applied to the raw data before model training. These transformations are often applied in two stages: the (optional) first stage performs an analysis of the data to derive required statistics for the transformations deriving necessary statistics for the transformation; the second stage uses these statistics to apply the transformations. As an example, consider the common z-score transform for numerical features: the first stage computes the mean and standard deviation of the feature values, and the second stage uses these two metrics to normalize each feature value. In general, the second stage is embarrassingly parallel and can thus easily scale to large datasets. The first analysis stage, however, is much more expensive as it requires potentially expensive aggregations over the data (e.g., sorting a large space of video ids based on frequency), which

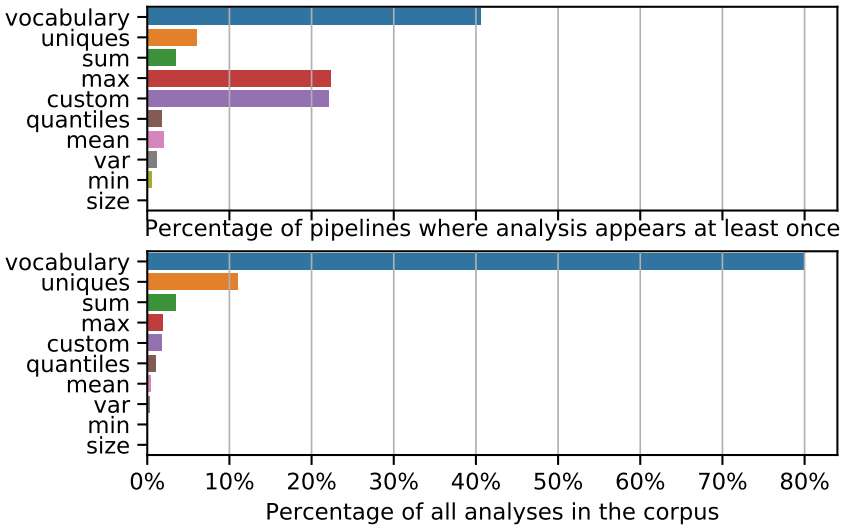


Figure 4.4: Analyzer Usage

are not embarrassingly parallel. We thus focus on this analysis stage in what follows, since that is where we see opportunities for data processing-related optimizations.

Figure 4.4 shows the different types of analyses applied in the first stage of feature transformations. The "vocabulary" analyzer performs the aforementioned truncation of a sparse feature into a smaller numerical domain. As an example, given a feature that contains text tokens (e.g., words), this analysis computes the top-K tokens based on frequency and then maps the features to the numerical domain $[0, K]$. The other types correspond to more straightforward analyses over numerical features (e.g., min, max, and so on). Finally, "custom" refers to a black-box analysis (essentially, a UDF) that is pipeline-specific and tailored to the business logic of the corresponding ML task. Pipeline authors resort to these UDF-style analyses when their features require more complex handling than what TFX offers out of the box.

Figure 4.4 shows two views of the same data. At the top, we show the percentage of pipelines that reference each analysis at least once, which indicates the relevance of these analyses across the pipelines in our corpus. The bottom view shows the total usage of these analyses across all traces and indicates the frequency of their usage in production. Both views confirm the prominence of vocabulary computations over categorical features. This becomes even more pronounced when looking at the actual usage in traces (bottom view). We observe that custom analyses are also used in several pipelines, although their total usage is much lower in the bottom chart. Our hypothesis is that such UDF-based analyses are more relevant for experimental pipelines that have a short lifespan and less activity, whereas canonical analyses provide good coverage for "steady-state" pipelines.

Both views in Figure 4.4 confirm the prominence of vocabulary computations over categorical features, which provides an interesting opportunity for the data management community, in two respects. First, this computation is a top-K query over an aggregation of the data where K can be very large, e.g., values of K from hundreds of thousands to millions are not uncommon. It is interesting to consider how these queries can be optimized for different representations of the data. Second, the choice of K has non-trivial implications on model quality and performance. A higher K implies better coverage of "important" sparse values, which in some cases results in significant improvements in accuracy. At the same time, since this mapping becomes part of the model, a higher K implies larger model sizes (to store the mapping) and perhaps higher processing time (to perform the mapping). To make an informed choice, the model developer needs to understand the tradeoffs for different values of K in terms of these dimensions, and this, in turn, introduces an interesting data analysis problem that may be amenable to techniques from data management (e.g., approximate query answering for this class of queries).

Model Diversity. We next examine the type of models used across pipelines. Doing so helps us characterize the diversity of training methods in production. Moreover, the choice of model architecture affects the selection of other operators and hence other characteristics of the pipeline. Figure 4.5 shows the usage of different architectures as a percentage of all models in our corpus. As we can see, 64% of the Trainer runs use deep neural networks

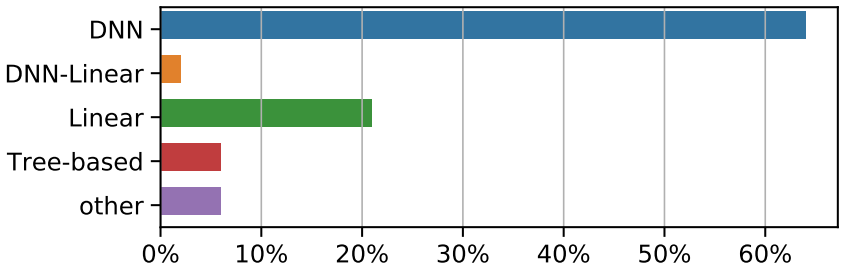


Figure 4.5: Percentage of Trainer runs with each model type

(DNNs), with an additional 2% that use a combination of DNNs with linear models. A smaller percentage of pipelines employ linear models and tree-based methods. Finally, a small fraction of the pipelines deal with specialized tasks requiring ensemble models or custom methods that we lump in the "other" slice.

It is interesting to relate this breakdown to recent papers that embed ML techniques inside database management systems and thus aim to jointly optimize data access and model training [104, 76, 142, 89]. A focus on a specific class of models (e.g., linear or DNN models) is still relevant in practice and can cover a significant fraction of production ML workloads. However, this would leave on the table a much bigger fraction of pipelines outside of the selected class that could also benefit from optimized systems. Moreover, the diversity shown in Figure 4.5 is an indication that ML practitioners need access to a wide range of choices, and hence are likely to "outgrow" a system that offers a few choices for model architectures.

4.3.3 Resource Consumption

Finally, we turn our attention to the composition of pipelines in terms of the different operators and their corresponding resource footprints.

Figure 4.6 shows the different types of TFX operators present in our traces and the corresponding percentage of pipelines using these operators. Here we group them in terms of their high-level functionality in the pipeline: data ingestion; data analysis and validation; data pre-processing; training; model analysis and validation; and model deployment. Perhaps unsurprisingly, the most common operators include data ingestion, data pre-processing, training, and deployment, with training and deployment in 100% of the pipelines since our corpus focuses on ML pipelines that support production applications. It is also worth noting that about half of the pipelines employ data- and model-validation operators, which essentially block the deployment of the trained models if there are errors in the data or if the model metrics are not sufficiently good, respectively [28, 52]. These operators act as safety checks and are common in pipelines that trained models used in downstream production systems.

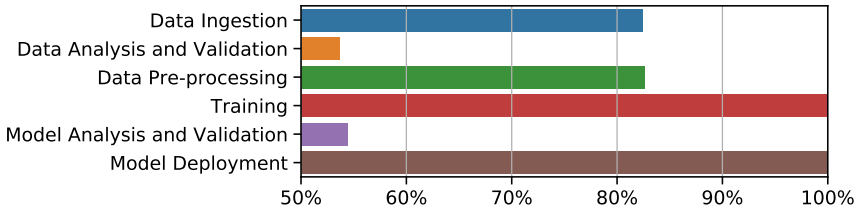


Figure 4.6: Percentage of pipelines with different operators.

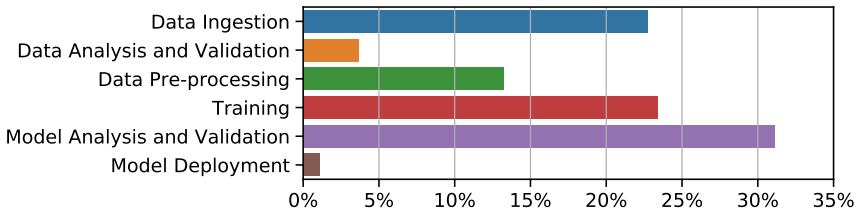


Figure 4.7: Compute cost of different operators.

Figure 4.7 shows a different breakdown of operators based on their resource usage. For each group, the figure shows the total compute cost as a percentage of the overall compute cost by all groups. Perhaps surprisingly, we see that the training operators account for less than a third of the total computation. This finding goes against the conventional picture that ML is mostly about the training algorithm. In contrast, our results indicate that production ML involves more steps that are also more costly, e.g., the data/model analysis and validation operators account for ∼35% of the total compute cost of the pipelines and are more expensive than training. Note that like data analysis and validation, model analysis and validation also boils down to traditional data processing operations, since it involves the computation of model metrics (e.g., average loss, area-under-the-curve) on slices of the input data, i.e., group-by queries with a model-driven aggregation per group. The figure also shows a significant cost for data ingestion (∼22%). This happens because, in many cases, TFX initiates a "hermetic" copy of each data span from the external data source, along with shuffling/partitioning of the examples to different splits (training/testing/eval) for the downstream model. This cost can be avoided for data sources that provide snapshot-based access with randomization guarantees for the individual examples.

Besides showing the cost of operators beyond training, the breakdown in Figure 4.7 reveals one more point: pipeline failures can be costly in terms of resources. Specifically, ML pipelines have several failure points corresponding to the different operators, e.g., the pipeline may stop because the data contains errors, or because the training code has faults, or because model validation failed. In turn, each failure point may occur after the successful completion of upstream operators, several of which can be costly based on our analysis. In

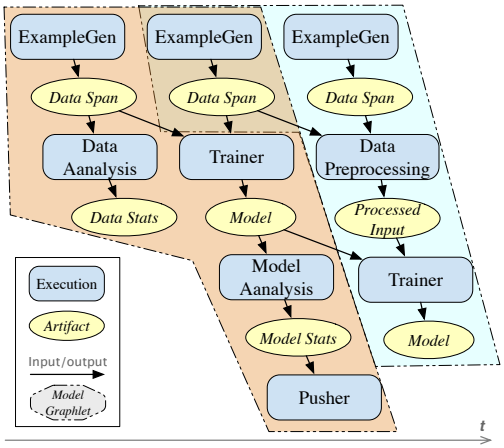


Figure 4.8: A model graphlet example

other words, failures are not cheap and there is an upside to preventing them or dealing with them proactively. Moreover, artifact caching and reuse, when feasible, can cut down significantly on the cost of different stages. One example is restarting a failed pipeline due to errors in the training code—since the data has not changed, it should be feasible to reuse any data transformations and thus avoid the cost of re-analyzing the data. We can use the costs in Figure 4.7 (in conjunction with failure probabilities) to determine optimized materialization policies, identifying where it might be most valuable to cache artifacts, e.g., after pre-processing, training, or model validation, as was done in recent work [209].

4.4 Fine-Grained Graphlet Analysis

The previous section presented a coarse-grained analysis that focused on aggregated characteristics across pipelines. In this section, we dial up the resolution of the analysis to investigate fine-grained characteristics within each pipeline, and in particular, the cadence and characteristics of model training and deployment—which, as stated earlier, can happen several times within a trace.

4.4.1 Model Graphlets

Recall that pipelines can be continuous, in that they consume a stream of incoming data spans to train and produce fresh models. Hence, a single pipeline trace may contain multiple executions of the Trainer operator (and associated downstream and upstream operators) on overlapping inputs. For instance, when using a rolling-window on the input data spans to update models, a data span might contribute to several models; when using warm-start during training, a generated model artifact might be used as an input to other Trainer executions. As a result, the pipeline trace is intricately tied to the complexity of ML in

practice, and it is not uncommon for the trace to have a single large connected component that encompasses all nodes, which in turn makes it difficult to examine the finer-grained characteristics of the pipeline (e.g., Figure 4.2).

The issues of inter-connectedness and size of provenance graphs have similarly emerged in different domains, wherein techniques such as user views, segmentation, and aggregation have been explored to transform the graphs to usable or interpretable ones [26, 3, 129, 134]. We adopt a similar approach but we leverage the semantics of production-ML operators and connections between them. Specifically, we segment the trace graph into several subgraphs, so that each subgraph represents a single (logical) end-to-end pipeline run related to an individual model. We provide some intuition for this definition in the context of an example below. We refer to such a subgraph as a *model graphlet*, or graphlet for short.

Datalog Queries for Graphlet Segmentation. Formally, a trace is a directed acyclic graph $G(V, E)$, where $V = \mathcal{A} \cup \mathcal{E}$ includes all artifacts $\mathcal{A}$ and executions $\mathcal{E}$ over time, and E is the union of all input $(\mathcal{A} \times \mathcal{E})$ and output $(\mathcal{E} \times \mathcal{A})$ relations between those operator executions and pipeline artifacts. Given a trainer execution $n \in \mathcal{E}$, a corresponding graphlet $g_n \subseteq G$ is a subgraph of the trace, where its nodes $g_n(V)$ include n and can be derived by the following datalog query:

```
g(V) :- E(V, X), g(X)
g(V) :- g(X), E(X, V), NOT sc(V)
```

where *sc* is a predicate excludes descendant executions that are on the path to other trainer executions. In our context, the *sc* is either Transform or Trainer executions as shown in Figure 4.1(b).

Example. Figure 4.8 shows the graphlets extracted on a sample trace. Intuitively, each graphlet corresponds to a model and captures the subgraph of the trace that is relevant for the generation of the model. Note that the first model is used to warmstart the Trainer for the second model, yet this edge is a "cut" between the two graphlets. The motivation for graphlets is to create smaller graphs of bounded complexity, so that we can analyze the corpus at the granularity of individual models. The graphlets also include data-analysis artifacts, so that we can analyze data-related metrics (reuse and similarity) across graphlets.

From this point onward, we recast our analysis on the set of graphlets that we extract from the corpus. In total, this gives rise to 450,000 graphlets.

	[0, 0.25]	(0.25, 0.5]	(0.5, 0.75]	(0.75, 1]	μ
Jaccard	30.2%	8.2%	4.4%	57.3%	0.647
Dataset	89.7%	0.3%	0.1%	9.9%	0.101
Avg Dataset	87.3%	5%	3.1%	4.6%	0.092

Table 4.1: Similarity metrics for consecutive model graphlets; percentage of consecutive graphlet pairs in each similarity range, along with the mean similarity.

4.4.2 Data Change across Graphlets

Data characteristics, and the evolution thereof, can have a significant impact on ML model performance [156]. We thus begin our analysis by investigating data evolution across graphlets of the same pipeline. Specifically, we seek to understand to what extent data is reused across different graphlets and the rate at which data distribution shifts over time. Answering these questions can provide useful insights related to ML data management, e.g., whether materialization of intermediate data transformations and incremental computation can be useful for ML pipelines. The analysis that follows uses the notion of *consecutive graphlets*. Two graphlets g and g' from the same pipeline are said to be consecutive if and only if the corresponding trainer executions are adjacent in chronological order, using the timestamps of the trainer executions for ordering.

4.4.2.1 Reuse

We quantify data reuse with the Jaccard similarity between the data spans of consecutive graphlets g and g', i.e., $|\mathcal{I}(g) \cap \mathcal{I}(g')|/|\mathcal{I}(g) \cup \mathcal{I}(g')|$, where $\mathcal{I}(g)$ is the set of input data spans in g.

The first row in Table 4.1 shows the histogram of the similarity values in the corpus. The high value at (0.75, 1]—57% of all pairs of consecutive graphlets—is due to the fact that many pipelines train multiple parallel models on the same inputs for A/B testing. Moreover, consecutive graphlets can correspond to retrainings on the same data after the pipeline author changes other details of the process, e.g., the training algorithm or feature transformations. The mean similarity of .647 indicates that, on average, graphlets share two thirds of the data spans. Moreover, several consecutive graphlets have a $> 80\%$ similarity of their inputs. These results point to interesting optimization opportunities on data preparation for training (e.g., data pre-processing, data transformation, or data validation). Specifically, we can leverage this overlap and employ techniques from incremental data computation and view maintenance to efficiently perform the data preparation steps for a new training run. Indeed, most of the data analysis operators in Section 4.3.2 lend well to incremental view maintenance techniques [96].

4.4.2.2 Dataset Similarity

The previous analysis focused on the reuse of the *same* data spans across graphlets. However, what if the data spans are different but their contents have similar distribution? Answering this question can help us identify further opportunities for reuse and optimization, albeit of a different nature. For instance, if two consecutive graphlets have data inputs with the same distribution, then some aspects of data preparation can be reused even if the data spans are different, e.g., the second graphlet can reuse the vocabularies of the first graphlet over the same categorical features (see also Section 4.3.2). One "extreme" optimization is to skip training altogether, given that the data distribution is the same!

How can we measure dataset similarity between consecutive graphlets? As noted above, the input to a graphlet can include multiple data spans. However, we do not have access to the actual data in each graphlet, since this data is not part of our corpus and it might also be siloed for privacy reasons. Instead, the only information we have about each data span is summary statistics for the set of numerical and categorical features in the span (see Section 4.3.2). These limitations make it hard to reuse existing data-similarity metrics as is, which either require full knowledge of the data, not just summary statistics, or do not handle comparisons of sets of data spans.

We thus adapt existing metrics to quantify dataset similarity in our specific setup. We do not claim novelty of this metric; rather, our goal is convey that this adapted metric provides reasonable and intuitive results for quantifying span-content similarity. A formal investigation on similarity metrics for this type of data is beyond the scope of our work. Next, we first give a high-level overview of our proposed similarity metric to establish intuition and then define the metric formally in detail.

The proposed metric has a layered formulation. First, we consider how to *compare a pair of data spans using summary statistics* on the constituent features. We treat each span as a set of features and use Earth Mover's distance [168] to compare the two sets, with a feature-to-feature similarity that is based on a hashing scheme on probability distributions [121]. Second, given this span-pair metric as a building block, we introduce a metric to *compare two sequences of spans* that come from different graphlets. The metric aligns the two sequences by position and computes the sum of span-to-span similarity values normalized by the length of the longer sequence. We use sequences instead of sets to model the sequential visitation of data for certain ML algorithms. If this aspect is not important for a different workload/system, then it is possible to use other metrics such as maximum bipartite matching.

We provide a detailed description of the metric below for readers who might be interested in applying the metric to other use cases.

Comparing a pair of data spans using summary statistics. A data span D comprises n features $\{f_1, f_2, \ldots, f_n\}$, with each feature being either *categorical* or *numerical*. For data privacy concerns, the raw feature statistics are not recorded. Instead, for a numerical feature, we have the discrete distribution of the feature values over 10 equi-width bins, with the range rescaled to $[0, 1]$; for a categorical feature, we have the count of the top 10 most frequent terms, the count of the unique terms, and the total number of datapoints, with all terms anonymized. First, we transform the term frequencies for categorical features into a probability distribution by sorting the normalized term frequencies in the descending order and setting the bin width to be $\frac{1}{N}$, where N is the number of unique terms, to obtain a discrete distribution over $[0, 1]$. For terms outside of the top 10 most frequent terms, we distribute the remaining mass evenly over the $N - 10$ bins. Note that we introduce sorting so that we can capture similarity over the shape of the distribution, independent of the obfuscation of the actual categorical values.

Standardizing feature representation across numerical and categorical features allows us

to devise a single unified feature similarity metric for both types of features. For efficiency, we use a locality sensitive hashing scheme called S2JSD-LSH designed for probability distribution [121] to compute an integer hash value $h(f_i)$ for a feature f_i. The similarity $s(f_1, f_2)$ between two features f_1, f_2 with feature names n_1, n_2 is then defined as

$$s(f_1, f_2) = \alpha \cdot \mathbb{I}(h(f_1) = h(f_2)) + \beta \cdot \mathbb{I}(n_1 = n_2) \tag{4.1}$$

, where $\mathbb{I}$ is the indicator function. We only compare features of the same type; the similarity between a numerical feature and a categorical feature is always 0.

Comparing two sequences of spans. Given two data spans $D_1 = \{f_i\}$ and $D_2 = \{f'_i\}$, where the f_i's and f'_i's are the features in the respective datasets, the dataset similarity $S(D_1, D_2)$ is defined as the Earth Mover's distance (EMD) [168] where features are treated as clusters with equal weights within each dataset and the distance between clusters is the feature similarity defined in Eq(4.1). Since it is difficult to determine a one-to-one mapping between the features given that the feature names are anonymized, we use EMD to capture the uncertainty in the mapping. This metric enjoys the property of being symmetric and having a range of $[0, 1]$. Moreover, $S(\varnothing, D) = 0$ where $\varnothing$ is the empty data span, and $S(D, D) = 1$.

We extend $S(D_1, D_2)$ to compare two sets of data spans as follows. Let $\mathcal{D} = (D_1, D_2, \ldots, D_n)$ and $\mathcal{D}' = (D'_1, D'_2, \ldots, D'_m)$ be the sets of input data spans for the two graphlets, where the indices within each set denote order by time of ingestion. We define the overall data similarity as

$$\mathcal{S}(\mathcal{D}, \mathcal{D}') = \frac{1}{\max(n, m)} \sum_{i=1}^{\min(n,m)} S(D_i, D'_i) \tag{4.2}$$

Simply put, $\mathcal{S}$ is the sum of pairwise data spans matched by ordinal position, normalized by the maximum number of spans between $\mathcal{D}, \mathcal{D}'$. We match by ordinal position instead of by identity of the data spans because ordinal positions can better handle rolling windows of data spans that may or may not overlap. Furthermore, certain training algorithms visit data spans sequentially ordered by ingestion timestamp. Like S, $\mathcal{S}$ is also symmetric and falls into $[0, 1]$.

The second row in Table 4.1 shows the quartiles of data-similarity values over all pairs of consecutive graphlets in the corpus. Similar to Jaccard similarity (first row), dataset similarity is bimodal at the first and last quartiles. However, the trend is reversed, due to the fact that data spans common in both sets may not be paired up in to ordinal position matching. As explained above, this is by design to account for cases of sequential data visitation in training. The third row in Table 4.1 shows the quartiles of data similarity when the latter is averaged over all graphlets in the same pipeline. Compared to the second row, we observe a drop in the higher quartiles, which indicates that pipelines with a large number of graphlets have more dissimilar pairs. Put differently, long-running pipelines belonging to power users have higher data volatility, motivating the need for data validation to safeguard against data errors and drift.

4.4.3 Model Retraining and Deployment

In this section, we use the graphlet decomposition to examine the relationship between model (re)training and deployment in the pipelines. We measure the duration of a graphlet as the difference between the start of the execution with the earliest timestamp in the graphlet and the end of the execution with the latest timestamp. As our analysis shows below, the conclusion is that (a) there are many trained models that are not deployed, and (b) very likely they correspond to wasted computation, which in turn introduces an interesting optimization opportunity.

4.4.3.1 Training vs. Deployment

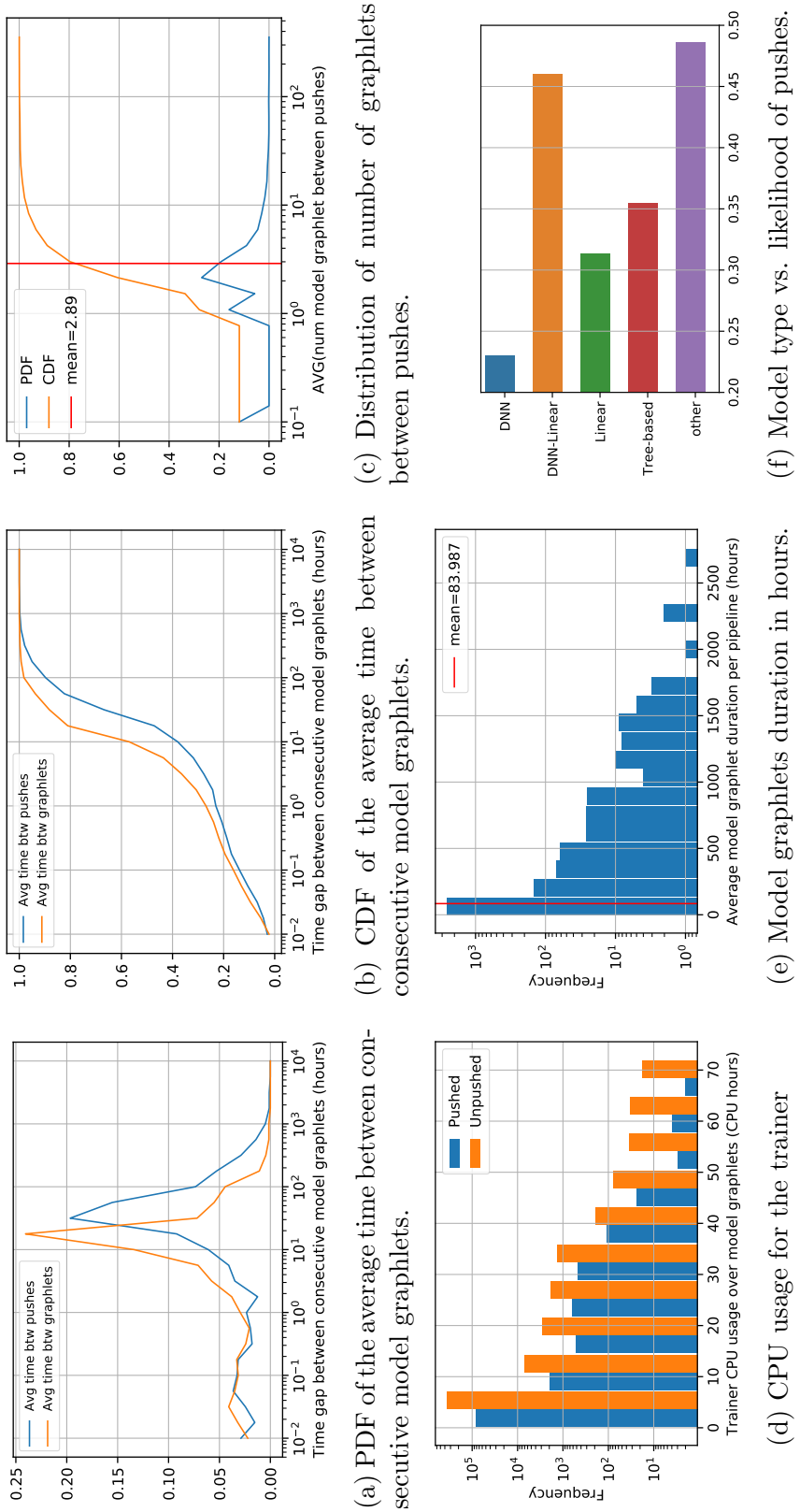
A model *push* marks the deployment of a trained model to a downstream service, thus making the model "visible" outside of the pipeline. In TFX, a push is performed via the execution of the Pusher operator. Note that a newly generated model may remain unpushed (e.g., due to failure to validate, or downstream throttling), in which case downstream services will keep using the last-pushed model. In this sense, a model push causes a model "refresh" for the downstream services.

Figure 4.9(a) shows the distributions of this time gap (in hours) for the two classes, while Figure 4.9(b) shows the cumulative view of the same distributions, both in log scale on the x axis. Immediately, we see that the two distributions have the same shape, but the mean is upshifted by ∼15 hours for the time between pushed graphlets. This is corroborated by the CDFs, which show that the time between 80% of all graphlet pairs is less than the median value for the time between pushed graphlets. These trends can have two possible explanations: 1) pushed graphlets tend to have longer duration than unpushed graphlets, or 2) pushed and unpushed graphlets have similar durations but are interleaved, thus widening the time gap between pushes. The second explanation has important implications for system optimization, motivating further effort to unearth the cause.

To further investigate the discrepancy between model training cadence and model push cadence, we plot the distribution of the number of graphlets between consecutive model pushes in Figure 4.9(c). We see that very few pipelines have no intervening unpushed graphlets between consecutive pushes, while most pipelines have between 1 to 10 unpushed graphlets between consecutive model pushes, with an average of ∼3. Furthermore, Figure 4.9(d) shows that unpushed graphlets have higher CPU usage for model training than pushed ones overall. These two facts together confirm that the second explanation presented above, namely the interleaving of pushed and unpushed graphlets with similar duration, is indeed the cause for the difference in distribution observed in Figure 4.9(a).

4.4.3.2 Model Freshness vs. Wasted Computation

The conclusion that pushed graphlets are interleaved with unpushed graphlets is a cause for concern for two reasons: 1) if the application requires model pushes to keep up with the ingestion of new data, i.e., the model must be trained and deployed before the the next



(a) PDF of the average time between consecutive model graphlets.

(b) CDF of the average time between consecutive model graphlets.

(c) Distribution of number of graphlets between pushes.

(d) CPU usage for the trainer

(e) Model graphlets duration in hours.

(f) Model type vs. likelihood of pushes.

Figure 4.9: Model graphlet analysis.

batch of new data is ingested by the pipeline, then a mismatch between the cadences of model training and model pushing implies an "unhealthy" pipeline; 2) if the application is tolerant of a slower model refresh cadence than data ingestion cadence, model training runs that do not lead to model pushes can be skipped to minimize wasted computation. In either case, the unpushed graphlets are potentially wasted computation.

While we do not have the requisite telemetry to precisely characterize when unpushed graphlets are indeed wasteful (e.g., unpushed models might be transitively useful by warmstarting models that do get pushed), our analysis results present quantitative evidence that the total amount of wasted computation is high. First, we observe that approximately 80% of all graphlets are unpushed. If none of the graphlets were used for warmstarting and there were no overlaps between graphlets, 80% of graphlets would account for ∼80% of overall computation since pushed and unpushed graphlets have similar CPU usage. To account for warmstarting and potential overlaps between graphlets, we can simply 1) remove graphlets belonging to pipelines with warmstarting, which account for 9% of total graphlets, and 2) remove all computation cost for operators that could potentially overlap, which accounts for $\sim 60\%$ of overall computation resource consumption, computed by removing the warmstarting pipelines from the results in Figure 4.7. Even with these generous assumptions the waste is still at $> 30\%$.

There is clearly an interesting question in whether we can identify the root cause of this inefficiency. However, further analysis shows that it is unlikely to find simple explanations. (We develop a more involved solution for the problem in Chapter 5.) Specifically, suppose that we attempt to explain the inefficiency through the following hypotheses: 1) rate-limited push: model pushes are configured to be at least a certain time apart, and models are trained faster than the predetermined push rate; 2) model types: certain model types can be more error prone and fail due to non-determinism with all else held equal; 3) data drift: shift in data distribution prevents models from passing validation checks; 4) code change: updating the code for the Trainer introduces system or model bugs.

To check (1), we compare the distribution of the model training time shown in Figure 4.9(e) with the distribution of the time gap between graphlets shown in Figure 4.9. While the average time gap between pushed graphlets is ∼40 hours, the mean model training time is 168 hours, making it highly unlikely that models are trained faster than the allowed push rate. This eliminates 1) as the cause of wasted computation.

To check (2), we observe from Figure 4.9(f) that the likelihoods of graphlets being pushed for different model types is highly variable. Moreover, all model types have a likelihood < 0.6, indicating that no single one is the culprit. However, it is unclear whether the low likelihood for some model types is due to the model type being error prone or other confounding factors.

Finally, to check (3) and (4), we compare the means for input data similarity, as measured by the metric introduced earlier, and code match (where 1 indicates match and 0 indicates no match) with the immediate preceding graphlet for the pushed and unpushed graphlets shown in Table 4.2. Overall, between consecutive graphlets, the input data similarity is 0.101, and the code stays the same 84.5% of the time. For both measures, we observe no significant difference between the pushed and unpushed groups. These findings suggest that

	μ_{pushed}	$\mu_{unpushed}$	μ
Input data similarity	0.109	0.099	0.101
Code match	0.838	0.846	0.845

Table 4.2: Model push vs. data drift and code change.

code change and data drift are not correlated with a graphlet not pushing a model.

4.5 Conclusion

To our knowledge, we presented the first ever large-scale analysis of production ML pipelines, based on a large corpus of pipelines from Google. Our analysis demonstrates the high complexity of these pipelines and the importance of operators other than training, in particular operators related to data preprocessing and analytics. Furthermore, we analyzed the cadence and characteristics of the generated models and characterized their relationship with respect to the input training data. The overall analysis revealed several points where techniques from data management can optimize different steps of these pipelines, including approximate query processing for determining the vocabulary size for categorical features and incremental view maintenance for computing features that depend on summary statistics in a data stream. Our analysis also revealed that unpushed graphlets represent a significant portion of the total pipeline computation, even though they are not likely to have an observable impact on subsequent models or downstream services. In the next chapter, we discuss an optimization to minimize wasted computation associated with unpushed graphlets.

Chapter 5

Improving Computation Efficiency in Production Model Deployment

As an immediate consequence of (and evidence for the value of) our analysis in Chapter 4, we identify a significant optimization opportunity involving preemptively skipping pipeline executions. Overall, there are many wasteful graphlets that neither deploy models to serve applications nor help warmstart subsequent model training. We show that such graphlets have significant resource costs. Moreover, we show that the root causes for the wasteful graphlets are varied; hence, it is difficult to come up with simple heuristic strategies to identify them. Instead, we leverage the dataset at hand and develop an ML-based solution—we train a model that uses the current state of the pipeline to predict whether the graphlet run will result in a deployed model prior to its execution. The prediction is used to adjust the execution of the graphlet in order to conserve compute cost. For instance, the pipeline scheduler may choose to down-prioritize or stall such graphlets until the pipeline owner intervenes and fixes the underlying issue(s). The model achieves high accuracy and allows us to save up to 50% of wasted computation, without compromising graphlet runs that deploy models. Beyond this direct benefit, this optimization is indicative of the opportunities to optimize ML deployments through a holistic analysis of the ML provenance graph, in addition to localized optimizations of individual steps (e.g., reducing the footprint of the trainer).

Next, we introduce the decision function we designed for this prediction task. Clearly, the decision function has to balance between different types of errors and their effects. A false negative, i.e., skipping a graphlet that would have resulted in a model push, compromises model freshness, while a false positive, i.e., running a model graphlet that does not result in a model push, contributes to wasted computation. *Model freshness* is measured by $\frac{n_{TP}}{n_{TP}+n_{FN}}$, where n_{TP} is the number of true positives and n_{FN} is the number of false negatives. We discuss later a method to explore the tradeoff between the two error types and show that we can eliminate nearly half of the wasted computation.

Data. To study this problem, we form a dataset by filtering the previous corpus to only include pipelines that do not warmstart model training with previous versions of the model. Unpushed graphlets are useful if they help warmstart subsequent model training (see Sec-

tion 4.4.3) and so we should not consider them as wasted computation. This leaves us with 2827 pipelines containing 420k graphlets in total. The dataset contains 80% unpushed graphlets and 20% pushed graphlets. To account for class imbalance, we use balanced accuracy to measure the fitness of decision functions.

5.1 Problem Statement

Let $\mathcal{G} = \{g\}$ be a set of model graphlets and $\mathcal{Y} : \mathcal{G} \to \{0, 1\}$ be the indicator function that evaluates to 1 for pushed graphlets and 0 otherwise. The objective of the waste mitigation problem is to find

$$\underset{\hat{\mathcal{Y}} \in \mathcal{H}}{\operatorname{argmin}} \sum_{g \in \mathcal{G}} L_m \left(\mathcal{Y}(g) \cdot (1 - \hat{\mathcal{Y}}(g)) \right) + L_w \left(\hat{\mathcal{Y}}(g) \cdot (1 - \mathcal{Y}(g)) \right) \tag{5.1}$$

where $\mathcal{H}$ is the space of decision functions that we will explore to find $\hat{\mathcal{Y}}$, an approximation for $\mathcal{Y}$, L_m is the loss function for model freshness that depends only on false negatives, and L_w is the loss function for wasted computation that depends only on false positives. Intuitively, model freshness is only affected by false negatives, where pushed graphlets are incorrectly predicted as `unpushed` and therefore prevented from updating the externally visible model; on the other hand, wasted computation can only be incurred by false positives where an unpushed graphlet was erroneously run without resulting in a refreshed model downstream.

The loss functions can be designed to prioritize either cost saving or model freshness. However, it is difficult, in practice, to determine the tradeoff *apriori* without getting a sense of the complexity of the dataset and the decision function. To overcome this challenge, one can use a single loss function L for both L_m and L_w to weigh the two types of errors equally and allow $\hat{\mathcal{Y}}$ to be a real-valued function that assigns likelihoods to the two labels, namely `pushed` and `unpushed`. Once $\hat{\mathcal{Y}}$ is found, the tradeoff between compute cost and model freshness can then be made post-hoc by setting a specific threshold on $\hat{\mathcal{Y}}$ to produce a binary decision function. Another convenient consequence of setting $L_m = L_w$ is that the problem becomes a standard binary classification problem that can be solved using standard approaches.

5.2 Limitations of Simple Heuristics

The analyses from Sections 4.3 and 4.4 in Chapter 4 surface numerous candidate signals that can be used to solve the binary classification problem. We experimented with a few simple heuristics for solving Equation 5.1 derived from our findings.

Model type. Since Figure 4.9(f) shows that model push rate is correlated with the model type, we created a simple heuristic that predicts `pushed` if the model type in the graphlet has an average push rate higher than the push rate averaged over all model types. This heuristic yielded a balanced accuracy of 0.599.

Input overlap. The input overlap heuristic predicts `pushed` if the Jaccard similarity (from Section 4.4.2.1) with the previous graphlet is > 0.65, the average across all graphlets. This heuristic yielded a balanced accuracy of 0.580.

Code match. Finally, as a baseline, we created a heuristic that predicts `pushed` if the code does not change from the previous graphlet. The code match attribute is shown to be low-signal by the results in Table 4.2. This heuristic yielded a balanced accuracy of 0.517.

The best handcrafted heuristic (model type) achieved a balanced accuracy of 0.6. The large search space of heuristics, on top of the low performance of heuristics we handcrafted, motivates the machine learning approach to automatically utilize complex signals for decision making.

5.3 Machine Learning Based Approach

We approximate the decision function $\hat{\mathcal{Y}}$ in Equation 5.1 by training a supervised ML model. Each graphlet in the training data is labeled as `pushed` or `unpushed` as described above. For each graphlet, we create features based on its structure and associated metadata, mostly relying on the insights discussed in Sections 4.3 and 4.4 in Chapter 4. We also introduce features involving the immediately preceding graphlets in order to capture temporal signals such as data-span similarity.

5.3.1 Features

We partition graphlet features into four categories described below. The first two categories (shape and model information) are features extracted from the graphlet itself. The remaining two categories (input data and code change) are history-based, i.e., they are derived by comparing a graphlet with a window of graphlets that immediately precede it. A distinct feature is created for each ordinal position in the window, e.g., for a window size of three, `code_change_1`, `code_change_2`, and `code_change_3`, are created to indicate whether the code in graphlet g has changed compared to those that are 1, 2, or 3 graphlets prior to g.

Graphlet shape. Shape features include the count of executions corresponding to each operator, as well as the average input and output count for each execution. We partition the operators into *pre-trainer* operators that can execute without the output of the Trainer, the Trainer, and *post-trainer* operators that validate the output of the Trainer for safety and quality. For example, the graphlet in orange in Figure 4.8 has 2 ExampleGen with on average 1 output, 1 Trainer with on average 2 input and 1 output, 1 Data Analysis and 1 Model analysis both with average input and output count both being 1, and 1 Pusher with on average 1 input. Note that to obtain the execution count for a particular operator, we need to run the graphlet up to that operator, incurring computation overhead to obtain these features. Recall that the additional cost to run the post-trainer operators is $\sim$30% of the cost to run the pre-trainer operators, as shown in Figure 4.7.

Model information. We include the model type, e.g. Linear, DNN, etc., as well as the model architecture for graphlets containing DNN models, as one-hot encoded features. Figure 4.9(f) indicates that model type and model pushes are correlated.

Input data. Input data-related features include both overlap computed using the input Jaccard similarity (Section 4.4.2.1), and dataset similarity (Section 4.4.2.2), between a graphlet g and the graphlets preceding g as history-based features.

Code change. A binary feature for whether the code versions for the Trainer operator match between a graphlet g and the graphlets preceding g, as history-based features. While previous results in Section 5.1 have shown that the code change as a standalone feature is low-signal, we include it in the feature set to explore potential interactions with other features.

We also experimented with pipeline-level features inspired by the findings in Section 4.3, such as the data transformations, average time between graphlets and average graphlet duration in a pipeline, but found no significant improvement in model performance.

5.3.2 ML Model Training and Testing

We split pipelines in the corpus at random into a training and a test set, such that the total number of graphlets belonging to pipelines in the training set is ∼80% of the total number of graphlets in the entire corpus, and the distribution of pushed and unpushed graphlets are roughly the same in the training and test sets.

We experimented with a large variety of models including DNNs and Gradient Boosted Decision Trees, as well as more interpretable models, such as Logistic Regression and Random Forest, using the Scikit-learn library [150], and found that Random Forest performed comparably with the more complex models explored by the Auto-ML tool. We report our results from using the Random Forest model below and discuss lessons learned from the model next.

5.4 Evaluation

We evaluate the models on both model performance and their ability to generate execution policies for reducing wasted computation.

5.4.1 Classification performance

As mentioned above, Random Forest is the most accurate among interpretable models and has comparable performance with much more complex models. We present the results for four variants of the Random Forest model:

RF:Input has all of the features except the graphlet shape features;

RF:Input+Pre has all of the features in RF:Input plus the graphlet shape features for pre-trainer operators;

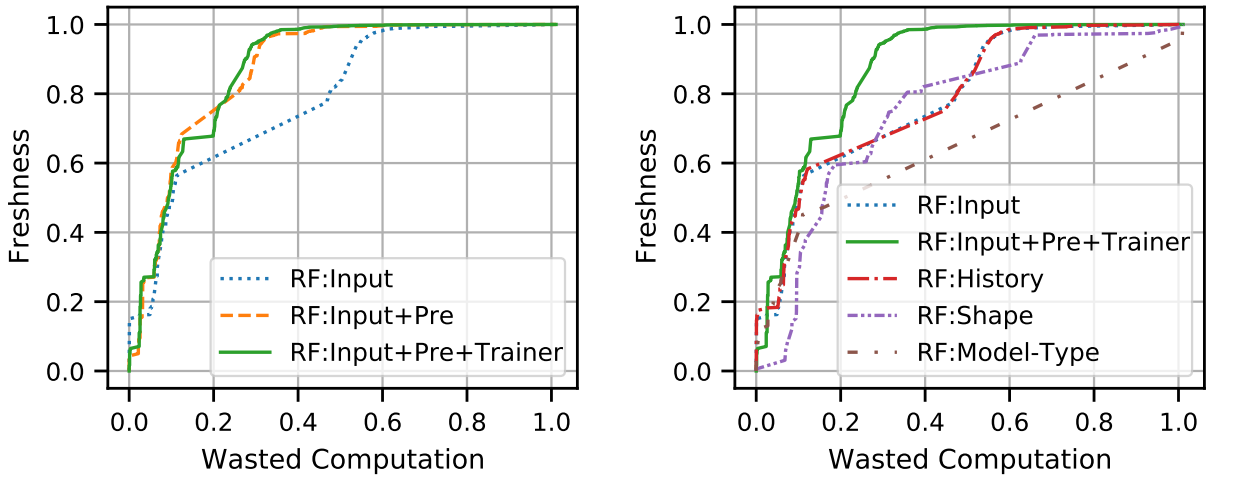
	Model	Balanced Acc.	Feature Cost
Random Forest	RF:Input	0.737	0.31
	RF:Input+Pre	0.801 (+9%)	0.53 (+71%)
	RF:Input+Pre+Trainer	0.818 (+2%)	0.77 (+ 45%)
	RF:Validation	0.948 (+16%)	1.00 (+30%)
Ablation	RF:Input	0.737	0.31
	RF:History	0.738	0.77
	RF:Shape	0.680	0.77
	RF:Model-Type	0.592	0.77

Table 5.1: Balanced accuracy for all model variants. The feature cost column indicates the compute cost to obtain the necessary features required by the models (rescaled to $[0, 1]$ with RF:Validation = 1).

RF:Input+Pre+Trainer has all of the features in RF:Input+Pre plus graphlet shape features for Trainers;
RF:Validation has all of the features in
RF:Input+Pre+Trainer plus shape features for post-trainer operators in the graphlet.

The four variants can be thought of as incrementally revealing more features about the shape of the graphlets as more operators are executed. They represent points in the pipeline execution that the system can intervene and abort execution to minimize wasted computation. For example, with RF:Input, the system can decide to abort as soon as the data is ingested; with RF:Input+Pre, the system has the option to abort right before model training. Note that RF:Validation is rather impractical in term of reducing wasted computation, as it requires all operators in the graphlet to be run to obtain the features. Also, the execution of validation operators can be highly correlated with whether the graphlet is pushed. We include RF:Validation as a proxy for an upper bound on prediction performance given complete (i.e., oracular) information.

Table 5.1 shows the model accuracy and the computation cost for the features used in each model, with the cost rescaled to $[0, 1]$ by dividing by the maximum cost attained by RF:Validation. Adding additional information leads to better model accuracies, however, the gain in accuracy does not grow linearly with the compute cost. Notably, from RF:Input+Pre to RF:Input+Pre+Trainer, the difference between the two models corresponds to shape features for the Trainer operator, which incurs a 45% increase in computation cost, for a measly 2% lift in model accuracy. All of the models significantly outperform the heuristics presented in Section 5.1, suggesting that there are complex interactions between signals that are hard to capture with heuristics looking at one signal at a time.



(a) Model freshness vs. wasted computation curves for Random Forest models.

(b) Model freshness vs. wasted computation curves for the ablation study.

Figure 5.1: Evaluation Results.

5.4.2 System Performance Improvement

We now examine how the trained classifier can help balance the tradeoff discussed earlier: skipping unpushed graphlets reduces wasted computation, but skipping pushed graphlets compromises model freshness in downstream applications.

Figure 5.1(a) depicts this empirical tradeoff between model freshness (y-axis) and wasted computation (x-axis) for the trained classifier. We compute this curve by doing a parameter sweep of the binary-classifier's threshold. Each threshold value results in specific false positive and true positive rates over the evaluation dataset, which we translate to values for wasted computation and model freshness respectively. As an example, consider the rates (0.65, 1) for a specific threshold value, which corresponds to a decision function that classifies correctly all the pushed graphlets and mis-classifies 65% of the unpushed graphlets (based again on our evaluation dataset). Since all pushed graphlets are identified correctly, model freshness is equal to 100%. Moreover, we can aggregate the total compute cost of the mis-classified unpushed graphlets, which in this case sums up to 70% of the total compute cost for these graphlets (equivalently, we recover 30% of the wasted computation). Hence, we map the decision function to point (0.7, 1) in Figure 5.1(a).

We highlight two important takeaways from Figure 5.1(a). First, we can eliminate 50% of all wasted computation without sacrificing model freshness at all. For a large-scale ML system like TFX, this amounts to a great deal of computation resources saved without affecting end-user experience. Second, model freshness does not drop substantially if we cut computation even further from 50% to 60%. However, following that, model freshness drops dramatically from 0.4 to 0, implying that we cannot hope to do much better than

removing 60% of the wasted compute without significant sacrifice on model freshness. While RF:Input is cheaper than RF:Input+Pre and RF:Input+Pre+Trainer as shown in Table 5.1, it is also a much worse execution policy. It can eliminate at most 30% of the wasted compute before model freshness suffers from serious decline. On the other hand, RF:Input+Pre and RF:Input+Pre+Trainer can easily eliminate 50% of the wasted computation with no impact on freshness. Thus, for a frugal user who is tolerant of model staleness, RF:Input might be the preferred model. A user who has strict model freshness requirements might prefer RF:Input+Pre to help them minimize waste without sacrificing model quality. The negligible improvement in prediction performance by RF:Input+Pre+Trainer compared to RF:Input+Pre does not justify the 45% computation overhead to obtain the features. Thus, RF:Input+Pre+Trainer, despite leading in prediction performance, is not as effective from a cost saving perspective.

5.4.3 Feature Ablation Study

To better understand the impact of the different groups of features introduced in Section 5.3, we conduct a feature ablation study, where Random Forest models are trained and evaluated with a subset of the features. The balanced accuracies for the ablated models are shown in the last four rows of Table 5.1. RF:History contains both the input data features and code change features. RF:Shape contains only the counts for the operators excluding validators. Of the four groups of features, RF:Input, while being the cheapest in terms of cost to obtain feature values, achieves the best performing model. Interestingly, adding the code features to the input features, as is done in RF:History, has no effect on the model performance. RF:Shape performs significantly worse than RF:Input but better than RF:Model-Type, which achieved the same performance as the simple heuristic involving model type.

Figure 5.1(b) shows the freshness vs. wasted computation curves for models in the feature ablation study. We use RF:Input+Pre+Trainer as the baseline since it is the best performing Random Forest model. Of all the feature classes, model features are the least informative by a long shot. The fact that RF:Input and RF:History have identical performance serves to validate that code changes are uncorrelated with model pushes. The interweaving of the RF:Input and RF:Shape curves suggest that these two groups of features are predictive for different subset of graphlets. The ablation study shows that no single group of features captures most of the accuracy gains, suggesting that there exist complex interactions between the different groups of features driving the performance of the best models.

5.5 Conclusion

In this chapter we tackled one of the optimization opportunities discovered from the analysis in Chapter 4, namely the problem of wasted computation for pipeline runs that were automatically triggered by new data ingestion but failed to deploy updated models to downstream services. The trade-off in this problem is between the amount of wasted computation

saved by skipping pipeline runs that would not have resulted in updated models and the risk of compromising model freshness by skipping runs that would have led to a model refresh. We show that our proposed ML-based solution is able to eliminate up to 50% of the wasted computation without sacrificing model freshness, with the potential to save even more provided tolerance of low-level impact on model freshness.

The problem and the solution presented in this chapter highlight an interesting duality of the impact of automation on ML systems: while indiscriminate automation in the model retrain policy led to the problem of wasted computation, better automation driven by key data insights is also the solution. This raises the question of what role automation should play in the ML tooling landscape. To answer this question, we turn yet again to need finding in Chapter 6.

Chapter 6

Understanding the Role of Automation in Machine Learning Development

In recent years, automated machine learning, or Auto-ML[1], a new field targeted at increasing automation during ML, has witnessed a rapid rise in popularity, driving an explosion in self-proclaimed Auto-ML tools. Auto-ML holds the promise to make ML more easily accessible to users to employ for new domains and to reduce cumbersome manual effort when applying ML to existing ones. Auto-ML was initially introduced to automate model hyperparameter search. Over time, Auto-ML has evolved beyond that to include, as part of its goal, automation for other tasks in the ML workflow such as feature engineering, data cleaning, model interpretability, and model deployment. The ultimate promise of Auto-ML tools is to make ML more accessible by providing off-the-shelf solutions for users with less technical backgrounds.

In order for Auto-ML tools to effectively meet user needs, we must first understand what those needs are and what roles automation currently plays in the ML workflow. Towards this goal, we conducted an in-depth, qualitative study of sixteen Auto-ML users. We conducted semi-structured interviews with current users of Auto-ML, ranging from hobbyists to industry researchers across a diverse set of domains, to take a careful look at their use cases and work practices, as well as their needs and wants. We asked about their uses of ML, their experiences working with and without Auto-ML tools, and their perceptions of Auto-ML tools. Our approach enabled us to not only gain a better understanding of user needs with respect to Auto-ML, and the strengths and weaknesses of existing Auto-ML tools, but also to gain insights into the respective roles of the human developer and automation in ML development.

Our findings indicate that human developers are indispensable in ML development. Not only do humans excel at bringing in valuable domain knowledge, human intuition can also be

[1]Stylized as Auto-ML to avoid confusion with Google AutoML.

effective in filling in the blind spots in Auto-ML. Furthermore, human involvement is crucial for the effective and socially-responsible use of ML in real-world applications. Therefore, rather than attempting to automate human developers "out of the loop" as has been the objective of many Auto-ML tool builders, we advocate for a symbiotic relationship between the human developer and Auto-ML to integrate the human into the loop in the most productive manner. We, in fact, advocate that the moniker "Auto-ML" be discarded, because our evidence suggests that complete automation is infeasible; instead, these tools can be better thought of as offering mixed-initiative ML solutions.

Several prior studies have advocated for a human-guided approach to Auto-ML [110, 67, 197] and proposed design requirements. Rather than designing top-down, we build our case for human-AI integration based on bottom-up findings of work practices, allowing us to arrive at unique, specific insights that are difficult to develop without taking the perspectives of the practitioner into account. We make concrete recommendations on what functionalities tool developers should enhance while preserving existing benefits, and, more importantly, what roles of humans they should preserve rather than attempt to replace. Our recommendations are based on both the perception and usage of a range of users of state-of-the-art Auto-ML tools. We hope that these insights and design recommendations can guide future development of Auto-ML tools to expand the current focus on system and model performance to emphasize human agency and control.

The rest of the chapter is structured as follows: we discuss the relationships between our work and relevant HCI contributions and present state-of-the-art on Auto-ML tooling in Section 6.1; we describe our study methodology in Section 6.2; we present findings on the benefits and deficiencies of current Auto-ML tools as well as the respective roles of the human developer and automation in the ML workflow in Section 6.3; we discuss the design implications of our findings on Auto-ML tools in Section 6.4 and conclude in Section 6.5.

6.1 Related Work

Auto-ML systems are often aimed at developing an end-to-end ML workflow or model, unlike other human-in-the-loop ML tools that are more focused on specific parts of the ML workflow, such as collection of training examples, model debugging, and model interpretation. Our work builds on multiple areas of prior work: existing Auto-ML systems, human-centered ML work practices, and human-in-the-loop ML tools.

6.1.1 Auto-ML Systems

The current landscape of Auto-ML tools is fast-growing and diverse. Existing Auto-ML offerings can be categorized into three groups: open-source software, hosted commercial solutions offered by cloud providers, and enterprise solutions offered by companies dedicated to developing Auto-ML platforms. Cloud-hosted Auto-ML solutions exist as part of a larger ecosystem of tools in the cloud, whereas enterprise solutions are standalone and therefore

must either provide end-to-end support or integrate with external tools, resulting in appreciable differences in the user experience.

As introduced in Section 1.1 in Chapter 1, a typical ML workflow can be partitioned into three stages: *data preprocessing*, *modeling*, and *post processing*, and Auto-ML solutions provide varying levels of support for each of these stages. Below we offer a brief overview of tools in these three categories and comment on their support for the three stages in the ML workflow.

Open-Source Software (OSS). Auto-ML tools in this category include libraries such as Auto-sklearn (based on Scikit-learn) [60], TPOT (based on Scikit-learn) [107], and AutoKeras (based on Keras) [83], all developed in academic labs, as well as TransmogrifAI (based on Apache Spark) [187], AdaNet (based on Tensorflow) [39], Ludwig [133], and H2O [71] from industry. Of these tools, AutoKeras, AdaNet, and Ludwig are designed specifically for deep learning, tackling issues such as efficient neural architecture search, while the others are designed for traditional ML. Of the three categories of Auto-ML tools, OSS tools are the best at keeping up with cutting-edge ML research since many of the OSS tool developers are also involved in ML research. Overall, users of these libraries are afforded great flexibility since they can easily integrate custom code with the Auto-ML API, but they must provision their own computation resources. For the specific stages in the ML workflow, the Scikit-learn-based libraries are better suited for structured data and have better support for automated data preprocessing than the other libraries, while the Tensorflow and Keras-based libraries can support more complex models involving text and images. OSS tools generally lack on post-processing support, which involves evaluation, deployment, and monitoring of models.

Cloud Provider Solutions. The major players in this category are Google Cloud AutoML, Microsoft Azure Automated ML, and Amazon SageMaker Autopilot [68, 130, 8]. Solutions in this category differ from the previous category in three significant ways: 1) Since they are hosted, compute resources are provided and managed by the cloud provider, and users pay proportional to the amount of compute consumed during the process of Auto-ML; 2) They tend to be much more end-to-end and include model evaluation and deployment to help users derive business value from the models trained; 3) Some system internals are opaque to the user, who can only interact with the system at specific decision points. Overall, while cloud-hosted solutions tend to require less programming expertise to use, they are also *less configurable and transparent*. For example, Google Cloud AutoML, which boasts "more than 10 years of proprietary Google Research technology to help your ML models achieve faster performance and more accurate predictions", neither allows users to specify the type of models nor provides visibility into the model internals. Amazon SageMaker Autopilot, on the other hand, places a strong emphasis on visibility and control by allowing users to easily export the Auto-ML code and intermediate results into computational notebooks. All three providers offer no-code UIs for non-programming users, in conjunction with Python APIs. While Microsoft and Amazon enable additional customizability through their Python APIs, Google's APIs are solely designed for programmatic compatibility and offer no additional control or transparency. When it comes to model evaluation, Microsoft and Amazon provide

summaries of the models explored while Google offers only high-level information about the final model.

Auto-ML Platforms. As self-proclaimed "platforms", tools in this category tend to position themselves as turnkey, end-to-end Auto-ML solutions. They manage compute resource provisioning by integrating with either cloud providers or on-premise hardware infrastructure. Major players in this category include DataRobot [44] and H20 Driverless AI [71]. Compared to their cloud provider counterparts, solutions in this category tend to be more feature-complete, providing more technical support and customizability in each stage of the workflow. Since these solutions target business users, special attention is paid to the operationalization of the resulting model, including an expansive set of model interpretability and deployment options. Additionally, to address increasing concerns around data privacy and security, these solutions allow on-premise deployment instead of forcing users to migrate their data to the cloud.

Thus far, there has been little evaluation on how these three types of Auto-ML tools are used in practice. In this paper, we sought to understand the adoption of Auto-ML tools, their usage, and the current bottlenecks that users face with these tools. Our eventual goal is to identify design requirements and considerations for more effective collaboration between the human developer and ML/AI.

6.1.2 Human-Centered ML Work Practices

HCI research in Auto-ML has proposed a human-guided approach to ML [67, 110, 197], with the aim of balancing the trade-off between user control and automation. Human-guided ML builds on the premise that ML development should be a collaborative activity between human users (i.e., data scientists or model developers) and the machine, wherein users specify their domain knowledge or desirable model behavior at a high-level and the system performs some form of automated search to generate an appropriate ML pipeline or model [67, 110]. This work is related to a larger body of studies on ML work practices [87, 11, 213, 81, 79] of specific groups of practitioners, including software engineers using ML [11], non-expert ML users [213], as well as specific aspects of work practices, such as model interpretation [81] and iterative behavior in ML development [79], including the study in Chapter 2.

Existing work has explored the use of visualization to help users gain insights into the black-box process of Auto-ML and obtain higher control during Auto-ML [198, 199]. Another crucial step towards human-guided ML requires understanding the perspective of data science practitioners' and their attitudes towards Auto-ML systems [197]. In particular, Drozdal et al. study the perceptions of transparency and trust in Auto-ML and identify components within Auto-ML tools most likely to improve trust [53]. We expand upon this work by studying the role of humans and Auto-ML in the end-to-end ML workflow, as well as additional important user requirements, such as customizability, completeness, ease-of-use, efficiency, effectiveness, and generalizability.

These prior studies have largely focused on participants without Auto-ML experience. As a result, most of these studies elicited user feedback regarding Auto-ML by demoing new Auto-ML prototypes to study subjects. These prototypes have limited features, largely supporting only the modeling stage of the ML process. For example, the study by Drozdal et al. involves university students and uses hypothetical tasks to determine the relative value of components of Auto-ML in communicating trust, rather than centering on the real-world experiences of Auto-ML tool users. Our paper builds on this line of work by studying users who have worked with present-day Auto-ML tools in real-world applications to investigate how Auto-ML fits into their end-to-end ML workflow, as well as its limitations. Our methodology and choice of participants afford novel findings beyond existing studies on the matter of trust and transparency, such as the broader social context surrounding the use of Auto-ML in the real world, e.g., getting others in the organization to adopt the results from an Auto-ML model, or Auto-ML playing a role in reproducibility and institutional knowledge should the original user leave the organization. Moreover, we identify various ways hands-on human agency and control promotes trust—something left unaddressed by studies involving hypothetical Auto-ML interfaces.

6.1.3 Human-in-the-loop ML Tools

The idea of incorporating human knowledge into ML workflow development has been well studied in research on interactive tools for ML model interpretation. Interactive machine learning (IML) is a paradigm wherein human users train an ML model by manually evaluating and correcting the model result through a tight interactive feedback loop [58, 61, 160, 10]. IML often focuses on collecting user input for labels for training data in order to train and correct ML classification models. IML lowers the barrier to interacting with ML-powered systems by empowering end-users without ML expertise to interactively provide exemplar feedback to the system [10, 213] (e.g., recommendation systems can learn from the relevance feedback of approving/rejecting an item), often eliminating the need for the highly-technical feature engineering step [58]. In addition to IML, human-in-the-loop systems have also been developed for model debugging and verification [149, 9, 144], iterative model development [209], and model interpretation [6, 203, 183] often through an interactive environment and/or visualizations. We study the impact of such capabilities as reflected in present-day Auto-ML tools.

Prior work questions the validity of humans in the loop [188]. However, the study experiments only examined the effects of ML recommendations on human decision making after a model has been trained. Our study shows that humans are in the loop throughout the ML workflow even as Auto-ML attempts to automate away ML development. Along the way, humans can iteratively influence the directions of ML outputs via several touch points. Further research is necessary to understand the effects of mixing ML and humans in real-world settings.

6.2 Study Design

Our paper seeks to understand how users incorporate Auto-ML tools in their existing workflows and their perceptions of the tools.

6.2.1 Recruitment

Since, to the best of our knowledge, no prior work has focused on participants with real-world Auto-ML experience, we focused on users who have applied Auto-ML to real-world use cases across a broad set of application domains. We recruited participants by posting recruitment messages to relevant mailing lists and Slack channels (N=5), through personal connections (N=9), and by reaching out to users on social media who mentioned or were mentioned in posts about Auto-ML(N=2).

We invited participants to fill out a screening survey that asked whether they had experience using Auto-ML tools. Select participants either had experience using at least one of the tools that were listed in the survey (the list includes tools discussed in Section 6.1.1) or had used other tools that our team verified to be Auto-ML tools. The tool-based eligibility criteria was motivated by our pilot recruitment strategy where a large number of survey respondents expressed that they had general ML experience but were unfamiliar with Auto-ML, mistaking manual ML for "automated ML".

6.2.2 Participants

We interviewed a total of 16 participants who had prior experience applying Auto-ML in professional capacities. Information about each participant is shown in Table 6.1. Of the 16 participants, 14 were male (87.5%) and 2 were female (12.5%). Recruiting participants past March 2020 was difficult due to COVID-19, but we gathered a sample of users that spanned a diverse set of organizations and use cases.

Participants had, on average, 10 years of experience in programming and an average of 5 years of experience with ML. Participants spanned three continents and a diverse set of job roles and industries, from a product manager at a large retail corporation, to a director of commercial data science at a travel technology company, to academic researchers at universities. The tools that the participants used also varied from proprietary tools, to open-source software, to commercial solutions. To preserve participant anonymity, we report the category of the Auto-ML tool used but omit the identities of the specific tools used, as some tool-application combinations can reveal the identity of the participant.

6.2.3 Interview Procedure

We conducted semi-structured interviews with participants about their experience in using Auto-ML for real-world applications. Interviews were conducted from October 2019 to March 2020, either in person (N =2) or remotely (N=14). Each interview lasted for approximately

PID	Yrs of Exp (programming/ML)	Organization Domain (Organization Size)	Participant Role	Application	Auto-ML Tool Category
P1	30/5	Finance (M)	Data Science	Financial credit scoring	Commercial (PF)
P2	8/2	Healthcare (S)	Data Science	Object identification in videos	Commercial (CP)
P3	4/7	Retail (L)	Product Management	Fraud detection in e-commerce transactions	Commercial (PF)
P4	11/6	Social Network (L)	ML Engineering	Content curation and recommendation	Proprietary
P5	10/3	University (L)	Biomedical Research	Drug response prediction	Open Source
P6	10/4	Healthcare (S)	Data Science	Medical claims analysis for healthcare program referral	Commercial (PF)
P7	12/2.5	Finance (L)	Data Science	Fraud detection in financial transactions	Commercial (CP)
P8	5/4	University (L)	Neuroscience Research	Diagnostics and phenotype analysis with brain scan images	Open Source
P9	8/8	ML Software (S)	CTO	Enterprise document processing automation	Commercial (CP)
P10	20/8	Travel Technology (M)	Data Science	Personalized travel recommendations	Commercial (PF)
P11	6/5	Information System (S)	ML Research	Computer vision for brand logo detection	Open Source
P12	16/12.5	Educational Software (M)	Data Science	Content identification in educational materials	Commercial (CP)
P13	6/3	Electronic Product Manufacturing (L)	Research	Gesture recognition in sensor data from edge devices	Commercial (CP)
P14	12/5	Consulting (S)	Software Engineer	Nonprofit AI consulting for businesses	Open Source
P15	4/3	Pharmaceutical (L)	ML Researcher	Drug response prediction	Open Source
P16	2/1.5	Supply Chain (L)	Program Management	Supply chain procurement and risk-management	Proprietary

Table 6.1: Interviewee demographics, from the left: (1) Participant ID, 2) Participant's years of experience (Yrs of Exp) in programming and in ML, (3) Domain of their companies and organization size in parentheses with "(S)" corresponding to small, "(M)" to medium, and "(L)" to large, (4) Participant job title, (5) Auto-ML application, (6) Type of Auto-ML tool(s) used by the participant. "(CP)" indicates that the commercial solution is provided by a major cloud provider and (PF) indicates that the commercial solution belongs to the Auto-ML platform category introduced in Section 6.1.1.

one hour. The interview guide can be found in the supplemental materials in [212]. Every participant received a $15 gift card for compensation. The interviews were largely semi-structured and involved three main components:

- Participants were asked to describe their job role, organization, and ML use cases.
- We asked the participants about their experience in developing ML workflows (without Auto-ML) and the challenges they faced.
- We asked the participants about their experience using Auto-ML tools, including the features of the specific tool used and how they integrated Auto-ML in their ML workflows, and their perceptions of Auto-ML tools in general, including customizability, effectiveness, interpretability, and transparency.

6.2.4 Study Analysis

We audio-recorded and transcribed all but one interview, where the participant did not consent to being recorded. After completing all the interviews, we engaged in an iterative and collaborative process of inductive coding to extract common themes that repeatedly arose in our data. The three interviewers independently coded the data using Dedoose [46], an online tool for open coding, to map data onto these categories. The interviewers met weekly and discussed themes and concepts to clarify ambiguity in the codes and established consensus in a code book. Afterwards, we conducted a categorization exercise, wherein some of our initial categories included advantages and disadvantages of Auto-ML, perceptions of Auto-ML, workflow strategies with and without Auto-ML, Auto-ML desiderata, general ML challenges, and Auto-ML adoption decisions. We used codes to facilitate the process of theory development and refrained from calculating inter-rater reliability to avoid potential marginalization of perspectives [122].

6.2.5 Limitations

Our interviews were structured around the participants' experience with a single ML task. While this practice afforded concreteness to the discussion, we may have missed out on diversity of use cases. The interviews focused on work practices around Auto-ML, which allow us to gain understanding of the strategies participants employed to integrate Auto-ML into their data science workflow, but limited our time in studying any particular conceptual perception (i.e. transparency, interpretability) of Auto-ML in-depth.

We acknowledge that while our participant population comes from a diverse background (with respect to geographical location, organization size and type, expertise, use cases, and tool type), it may not be a representative sample of the overall Auto-ML user base. For example, the proportion of women in our sample (14 male, 2 female) is slightly lower than the gender ratio (15% women) in the data science profession referenced in the 2020 BCG report [55].

6.3 Results

In this section, we present our findings from the interviews. This section is organized as follows: we describe how we group the users and use cases into high-level categories in Section 6.3.1; we enumerate common tasks in ML workflows reported by the participants in Section 6.3.2; we present the benefits of Auto-ML perceived by the participants in Section 6.3.3, the deficiencies of existing Auto-ML tools that we believe can be addressed by system and UI improvements in Section 6.3.4, and the roles of the human developer that the Auto-ML tools must respect and preserve in Section 6.3.5. Note that while the functionalities presented in Section 6.3.4 and Section 6.3.5 are both absent or lacking in existing Auto-ML tools, the deficiencies as described in Section 6.3.4 have near-term solutions using existing techniques (we provide many such suggestions in Section 6.4), whereas the roles in Section 6.3.5 may one day become within reach of Auto-ML tools through fundamental breakthroughs in knowledge representation and programming paradigms. We believe that assuming the roles in Section 6.3.5 should not be the objective of Auto-ML tool developers until then.

6.3.1 User and Use Case Segmentation

A given participant's ML skill-set and use cases can potentially influence how they use and perceive Auto-ML tools. Therefore, we wanted to study the relationship between users' expertise and their behavior around and perception of Auto-ML. To facilitate the discussion on how contextual information serves to explain user behavior and sentiment, we categorize the participants and their use cases as follows.

6.3.1.1 User Skill Levels

We group participants based on their past experience with ML into the following three categories.

ML Innovators. Participants in this group have formally conducted ML research, either on fundamental algorithms (P4, P11, P14, P15) or the application of ML for scientific discoveries (P5, P8). They command deep understanding of the mathematical underpinnings of the ML models they use.

ML Engineers. Participants in this group are skilled and experienced ML practitioners with formal training in applied ML (P1, P2, P3, P6, P7, P10, P12, P13). They are intimately familiar with popular ML models as well as ML libraries and tools.

Novices. Participants in this group (P9, P16) have no formal training in ML. They are domain experts in non-ML fields.

	Prod. Apps	Prototype	Research	Novice	ML Engineers	ML Innovators	Data Preproc.	Modeling	Post Proc.	Transparency	Interpretability	Customizability	Generalizability	Completeness	Ease of Use	Effectiveness	Efficiency
Cloud Provider	2	3		1	4		★★	★★	★★	2.2	3	3	3.2	3	4.4	3	3.6
Auto-ML Platform	2	2			4		★★★	★★★	★★★	3.5	4.5	3.5	3.75	4	5	4.25	4.5
OSS		3	2			5	★★	★★★	★	3.6	4.2	4.2	4.4	3.6	3.8	4.2	4
Proprietary	2			1		1	N/A	N/A	N/A	2	3.5	3	4.5	2.5	4.5	4.5	4.5
									Avg.	2.825	3.8	3.425	3.9625	3.275	4.425	3.9875	4.15

Figure 6.1: Tools: Characterization of Auto-ML tools used by participants by category. The tool categories are the same as the ones presented in Section 6.1.1, with the addition of "proprietary" for anonymous in-house Auto-ML solutions. The first three columns contain the cross tabulation of tool categories and use case categories. The next three columns in blue contain the cross tabulation of the tool categories and user expertise levels. The following three orange columns indicate the level of support for each of the three ML workflow stages based on features offered (not based on interview results). The next eight columns contain the average Likert scores provided by our participants. Cells in green are above average.

6.3.1.2 Use Case Categories

Production Applications. Use cases in this category pertain to developing ML models that drive applications or decision making with significant impact. Examples include training models to assist in financial service decisions, building recommendation systems for content curation, and fraud detection. P1, P3, P4, P7, P12, and P16 have use cases in this category.

Prototype. Use cases in this category involve prototyping ML models for industry applications, which are lower stake than the production application scenario since the model performance has no direct impact on business metrics or end-user experience. P2, P6, P9, P10, P11, P13, P14, and P15 have use cases in this category.

Research. Use cases in this category involve building ML modelings for academic research, where model results are shared and examined in detail. Figure 6.1 shows the cross tabulation of the Auto-ML tools category with use case categories and user expertise, the level of support for each of the three ML workflow stages based on features offered by each category of tools, and the average Likert scores that our participants gave to their Auto-ML tools. The support ratings for "proprietary" tools are omitted as we do not have full visibility into the these tools. While Likert scores are commonly treated as ordinal data, they are treated as interval data in our setting due to the natural correspondence between the scores and quintiles. Thus, the average Likert scores are sound and meaningful.

6.3.2 Data Preprocessing, Modeling, and Post Processing Tasks

In this section, we focus on how practitioners integrate Auto-ML tools into the end-to-end ML workflow. As mentioned in Section 1.1, an ML workflow is commonly partitioned into

three stages: data preprocessing, modeling, and post-processing. In this section, we report the common patterns of how Auto-ML fits into the each stage. Figure 6.2 shows a complete set of tasks the participants reported for each stage of the ML workflow. We characterize these three stages as follows:

1. In the data preprocessing stage, users prepare the data for ML, performing tasks including data acquisition, cleaning, transformation, labeling, and feature engineering.

2. In the modeling stage, models are trained on the data prepared in the preprocessing stage.

3. After models are trained, the post-processing stage span a broad range of activities, including model evaluation, interpretation, deployment, and user studies.

6.3.2.1 Data Preprocessing

Overall, data preprocessing is a time consuming and primarily manual process for most participants, who need to write code for exploratory data analysis and data manipulation. On average, participants reported spending roughly 50% of their time on data preprocessing, and roughly 80% of all data preprocessing tasks were performed manually. While some participants expressed the desire for Auto-ML to provide more automated support for data preprocessing, others believe that data preprocessing relies on human intuition and knowledge that is impossible or at least extremely challenging to codify and therefore cannot be automated. In either case, participants agree that data preprocessing is a crucial component in the workflow due to the adage "garbage in, garbage out." We defer discussions on desired but missing automated data preprocessing features and the role(s) of the human to Section 6.3.4 and Section 6.3.5, respectively. Here, we report on the common tasks in the data preprocessing stage and whether each task is currently carried out by Auto-ML or human developers.

Data collection. The first task in data preprocessing is *data collection* [166] for most participants except P3, P5, P6, and P15, who were provided and limited to specific data sources. Others had the freedom to incorporate organizational data assets obtained from relevant stakeholders (P4, P7, P8, P9, P10, P11, P12, P13, P14, P16) as well as open datasets (P1, P2) from the web, which are much less commonly used. A specific data collection task worth noting is *data labeling*, where significant human attention is required to annotate examples with the desired target label. P9 and P13 reported performing labeling themselves because it provided them with intuition about the data and allowed for fast turnaround. P11 and P12, both of whom worked with unstructured data, relied on crowdsourcing for data labeling. Data labeling is inherently manual, although there are recent developments in systems to lower the manual effort for data labeling [16, 29].

Data Wrangling & EDA. Following data collection is data wrangling, including data cleaning and missing data imputation, formatting, and transformation. Exploratory data

		P6	P10	P1	P3	P7	P12	P2	P13	P9	P16	P4	P5	P8	P11	P14	P15
	Tool Type	PF	PF	PF	PF	CP	CP	CP	CP	CP	P	P	OSS	OSS	OSS	OSS	OSS
	Expertise	E	E	E	E	E	E	E	E	N	N	I	I	I	I	I	I
	Use Case	T	T	PA	PA	PA	PA	T	T	T	PA	PA	R	R	T	T	T
Data Preprocessing Tasks	Data Collection		M	M		M	M	M	M	M	M	M		M	M	M	
	DC: Data Labeling						M		M	M					M		
	Data Wrangling & EDA	M	M	M	M	M	M	M	M	M	M,A	M	M,A	M,A		M	M
	Feature engineering	M	M,A	A	M,A		M					M	A	A		M	
Data Preprocessing Tools	Python (pandas, numpy)																
	Python (ML libraries)																
	SQL																
	Other PL (R, C++/#)																
	Visualization tools																
	Other																
Modeling Tasks	Hyperparameter Tuning			A	A	A	A	A	A	A		A	A			A	A
	HT: Neural Architecture Search						A			A					A	A	
	Model Selection	A		A	A	A		A			A	A	A				
	Feature Selection		A		A				M			M	A	A			
	Inform Manual Development		A					A	A	A	A			A			A
	Business Req. to ML Objectives	M	M	M							M						
	Feature Engineering												A			A	
Modeling Tools	Custom Python/R																
	Scikit-Learn																
	Pytorch/Keras/Tensorflow																
Post Processing Tasks	Generate Reports	M	A	M,A	A	M	A	M	M	M	A	M	M	M	M	M	M
	Present to Stakeholders			M	M	M							M		M	M	M
	Model Export/Deployment	M	A		A		A	A	A	M	A	M		M		M	
	Fine tune/Validate predictions		M	A		A	M	M	M		M		M		M		
	Model interpretation	M	M	A	M								M				
	Monitoring	M	A		A							M					
	Model Management				A			A				M					
	Other		M						M								M
Post processing tools	Visualization Tools																
	Python (pandas, NumPy)																
	Deployment (Flask, Algorithmia)																
	Other (proprietary, W&B)																

Figure 6.2: Use cases: tasks performed and tools used by participants in each stage of the ML workflow. The three rows at the top contain metadata about each participants. "Tool type" corresponds to the tool types in Figure 6.1: PF = Auto-ML platform, CP = Cloud Provider, and P = Proprietary. "Use case" corresponds to the use case categories introduced in Section 6.3.1: T = Prototype, PA = Production Application, R = Research. "Expertise" corresponds to the user skill levels introduced in Section 6.3.1: E = ML Engineers, N = novices, I = ML Innovators. In each task cell, "M" indicates that the participant performed the task manually while "A" indicate automation. A green tool cell indicates that the participant has used the tool for the given ML workflow stage. Columns as ordered to cluster participants who use the same type of tools and have the same expertise levels.

analysis (EDA) is integral to all data wrangling tasks. All participants except P11, who worked with image data, reported that they spent time on data wrangling. In addition to Auto-ML tools used by P5, P8, P16 to help with data wrangling, the Python pandas [123] and numpy [189] libraries are the most popular choices for data wrangling, followed by SQL. Tools in the "Other" category in Figure 6.2 include domain specific tools, Spark, and proprietary tools.

Data cleaning, especially missing data identification and imputation, is a common task. While most Auto-ML tools handle missing data imputation, participants stated manual intervention is necessary to decide on the appropriate imputation technique based on context (P1, P4, P5, P7). Participants reported that they spent significant manual effort to format the data for ingestion by the Auto-ML tool (P2, P3, P8, P10, and P12.) Specialized data wrangling tasks included data anonymization (P3), time series test set generation (P10), and resampling for class imabalance (P16).

Feature engineering. Feature engineering and feature selection are among the most automated data preprocessing tasks. Most Auto-ML tools are capable of both simple feature engineering such as one-hot encoding [202] and building complex features involving multiple input signals. However, over half the participants who performed feature engineering manually because they felt existing Auto-ML solutions are incomplete or inefficient (P6, P10, P14) or they believed that human intuition and domain knowledge could not be replaced by automation (P3, P4, P12). Interestingly, P5 and P14 both reported repurposing, or *"misusing"* (P14) modeling capabilities for feature engineering.

6.3.2.2 Modeling

As one would expect, the most common tasks that participants used Auto-ML for during modeling are *hyperparameter tuning* and *model selection*. *Feature selection* is generally carried out by the Auto-ML tool as a byproduct of model training. Participants who perform feature selection all use Auto-ML for feature selection except P4, who manually selects features before model training because they work with petabyte-scale data, and P13, who deploy models to edge devices with limited memory.

An interesting use case for Auto-ML modeling is to *inform manual development* (P2, P8, P9, P10, P11, P13, P15.) For example, P13 reported that they use the Auto-ML tool as a quick check for data quality and to validate manual data preprocessing. They would perform data wrangling manually if the Auto-ML tool identified anomalies in the data. Auto-ML also empowers *exploration of unfamiliar models and hyperparameters* (P2, P8, P10, P13, P15, and P16.) Auto-ML results are often used as a *benchmark for validating manual model performance* (P4, P5, P9.) Many participants reported that they performed the same modeling tasks manually alongside Auto-ML to understand and verify the Auto-ML results and to correct any errors made by Auto-ML. The modeling tools in Figure 6.2 refer to tools for manual modeling.

6.3.2.3 Post-processing

Post-processing spans a large variety of tasks depending on the participants' use cases. Visualization is an integral part of many of the tasks below.

Generating reports and sharing results. The most common post-processing task is to generate a report of the model results and relevant model search history. Most platform tools automatically generate such reports, in the form of summary statistics, leaderboards, and other visualizations. P10 enthusiastically shared that their platform tool was able to auto-generate documentation for legal compliance thereby greatly reducing the manual overhead for governance. While all of the cloud-hosted Auto-ML tools also auto-generate reports and visualizations, it is interesting that many participants adopted manual approaches to modify the default reports. In addition to generating reports for their own consumption, a subset of the participants (P1, P3, P5, P7, P11, P14, P15) who had to share and explain their results to other stakeholders needed to present their findings in text documents or slides with human readable explanations.

Deploying Models. Model export and deployment is the second most common post-processing task. Participants who did not perform model deployment were either using the models to inform human decision making (P1), handed off the model to a separate Dev Ops team for deployment (P7, P11) or used model results solely for research findings (P5, P15). Automated deployment was only afforded to users of hosted Auto-ML tools. The reason for manual deployment for participants who used hosted tools include 1) the model for a financial application needed to be vetting for security (P7), 2) the Auto-ML tool did not support automated deployment (P6), 3) the application relied on complex logic to incorporate the model output (P9), and 4) the model directly impacted end-user experienced and required staggered roll-out with human supervision (P4). Tools for deployment included Flask, Algorithmia, custom Python, and proprietary infrastructure.

Validating and Interpreting Models and Predictions. The two main techniques for building trust in and understanding Auto-ML outputs are point queries on the predictions, feature importance, and holistic visualization of high-level model characteristics. Most hosted solutions focus on supporting point queries and feature importance via visualizations. Manual efforts in this category were mainly for 1) cross validating with manual modeling results (P3, P10, P11), 2) application domain specific checks (P6, P12), 3) field testing specific predictions (P13, P16), 4) custom feature importance computation (P5).

Miscellaneous. Less common, nevertheless noteworthy tasks included model versioning (P2, P3, P4), on-device user studies (P13, P15), debugging manual implementation (P15), and most interestingly, converting an ML model into a set of if-else statements for more predictable and interpretable inference (P10).

6.3.3 Benefits of Auto-ML

In this section, we present findings on the major benefits of Auto-ML from our interview study. As shown in Figure 6.1, ease of use, effectiveness, and efficiency are the highest-

rated qualities of Auto-ML tools by the interview participants. We present specific benefits reported by the participants that corroborate these ratings below.

Enables and empowers novices. To ML novices, the greatest benefit of Auto-ML is that it enables business users to nimbly use ML to inform business decisions without "*a massive engagement with multiple consultants in multiple different teams*"(P16). P16 believes that Auto-ML leads to "*the democratization of advanced analytics throughout business units for people that don't have experience doing that kind of work*", and this sentiment is echoed by P10:

> "It allows for ... citizen data science to become a reality with the proper governance controls and proper management in place. At the bank I worked at previously ... Auto-ML was becoming adopted ... for robotic process automation. ... Anyone who's analytically competent ... starts rolling with it immediately."

However, Auto-ML can be a double-edged sword for novice users—users who treat it as largely a black box find Auto-ML to be a great enabler, while curious users who attempt to look inside the black box can become distracted and suffer from choice paralysis. P9 reported that while they were able to achieve a few percentage points in model performance improvement, Auto-ML *increased* their development time, due to the fact that the Auto-ML tool exposed them to a large number of new model types that led to many lengthy manual explorations out of curiosity. In addition to avoiding distractions, treating Auto-ML as a black box also leads to the added benefit of standardization.

Standardizes the ML workflow for better reproducibility, code maintainability, knowledge sharing. Another benefit of the black-box nature of Auto-ML tools is that by having a predetermined search space that doesn't change, there is more standardization of the ML development process, leading to better comparisons across models, code maintainability, and effortless knowledge transfer. The need to search through a large number of model types, which are often implemented in different libraries, has prompted Auto-ML tools to create a standardized abstraction of the ML workflow decoupled from specific APIs and model types. As a result,

- different models are easily comparable using standardized, normalized metrics (P3),
- the amount of code needed to implement different models is greatly reduced (P3),
- models are more reproducible (P15, P3),
- latest ML research can be easily incorporated into existing workflows (P3, P13),
- model training requires less human intervention as there are fewer errors (P2)

For hosted Auto-ML solutions that generate extensive reports on the end-to-end process, Auto-ML serves as a self-documenting, reproducible knowledge sharing platform (P10, P3). This is especially beneficial in industry, as P3 points out:

> "Data scientists are expensive and very in demand, and people leave the job a lot and, and the algorithms change all the time. If I quit my job today . . . I could be like, here's all the history."

Prevents suboptimal results due to idiosyncrasies of ML Innovators. A unique benefit of Auto-ML applicable only to ML Innovators is its ability to prevent suboptimal model performance resulting from idiosyncratic practices by ML Innovators with extensive ML experience. P5 relayed instances where the Auto-ML tool found models that they never would have tried manually but outperformed the conventional choices they made. They therefore dubbed Auto-ML the "no assumptions approach" to ML development. P8 preferred the fact that the only factor driving the decisions made by the Auto-ML tool is "the statistical properties of the data" and not their own familiarity of specific model types, thus removing "bias" from the process.

Builds models more effectively and efficiently. The ability to *build better models faster* is a major benefit of Auto-ML tools reported by many participants, especially ML Engineers. Most participants reported that Auto-ML led to significant improvements in *efficiency*, the time for developing models, and a moderate increase in *effectiveness*, the performance of the final model obtained. On a five-point Likert scale, participants on average rated improvement in efficiency ($\mu = 4.1$) higher than improvement in effectiveness ($\mu = 3.88$). Some participants reported an order of magnitude reduction in development time (P10, P16). Improvements in effectiveness experienced by the participants were often incremental; thus, participants did not deem more accurate models in isolation as a compelling reason for Auto-ML adoption.

In addition to reduction in model development time, another dimension of efficiency is the ability to experiment with significantly more models in the same time it took for manual development. Even if the overall model performance does not change, exploring more models in itself provides downstream benefits. Many participants felt that they were much more productive because they were able to explore more models in the same amount of time. For P10, extensive experimentation also led to better insights into the features and models:

> "The team that built out the manual process were only able to review two or three different models. I was able to look at 50. I was able to report on more insights, more understanding of the variable inputs than they were in the same amount of time, more understanding around why the model performed the way that it did." (P10)

Enables rapid prototyping. Traditionally, incorporating ML into an application from scratch is a lengthy process involving many stakeholders. The substantial overhead of adoption, on top of the uncertainty about whether ML will improve the application behavior, deters many from ML adoption. Auto-ML has significantly lowered the barrier to entry by enabling users to build quick prototypes to gauge the feasibility and potential impact of ML, without the cumbersome process of setting up infrastructure and codebase (P9, P11)

and full integration (P12, P13), especially in industry settings. The caveat is that for many industry users, Auto-ML is used for rapid prototyping only but not full development, due to a lack of confidence in its performance (P4, P13)

> "[I use Auto-ML] just to prototype and see how it works. I don't use it within the system within my industry job, but I use it as a prototype system there just to see how easy this task is." (P4)

> "I would use Auto-ML first to try things out, to understand. It would be a good starting point for a new application. I am convinced that it would generalize, but if I want a particular performance number, then I am less confident about how it would perform." (P13)

Fosters Learning.

Users who were able to inspect the search history of the Auto-ML tools reported that they learned about new modeling techniques (P8), implementation of specific ML algorithms (P9), model architecture (P11, P14), model performance on specific types of tasks (P10), and model resource consumption (P15). These learning opportunities emerged serendipitously, as the users were validating the predictions and interpreting the models. However, for P2 who specifically sought to learn from their Auto-ML tool T (tool name anonymized to preserve participant privacy), the lack of transparency greatly hindered learning:

> "Getting the model out of T proved to be extremely challenging. It was like 94% accuracy 94% precision. And when we tried that we didn't see anywhere close to that. We tried to actually open up the model and see how it actually structured it, which was extremely challenging. It took a couple of hours, and then we learned that their structure was something that [they have written a paper on], so it was extremely hard to use ... It basically looks more like a black box."

Prior to their experience with T, P2 regarded T highly on account of the cutting edge ML algorithms that T claims to incorporate. However, due to their frustrations with the blackbox nature of the tool and the lack of offline reproducibility, P2 eventually abandoned Auto-ML and reverted back to manual ML development.

6.3.4 Deficiencies of Auto-ML

In this section, we present findings on the deficiencies of existing Auto-ML tools that can potentially be addressed via systems innovations. We discuss the design implications of our findings in Section 6.4. There are other limitations of Auto-ML tools that stem from the complex social and psychological implications of human-machine collaboration in ML workflows, and we discuss these limitations in Section 6.3.5.

Lacks comprehensive end-to-end support. Figure 6.1 shows that *completeness*, the extent to which Auto-ML covers the end-to-end ML workflow requirements, is the second lowest scoring rating. As evident in Figure 6.2, Auto-ML is currently used primarily for automating model training, requiring users to do the heavy lifting for both data preprocessing and, to a lesser extent, post-processing using other tools. This reality directly contradicts some claims made by Auto-ML tool developers. In the Auto-ML platform category:

> "[DataRobot] supports all of the steps needed to prepare, build, deploy, monitor, and maintain powerful AI applications at enterprise scale. [It] even automates model deployment, monitoring, and management."

> "[H2O Driverless AI delivers] automatic feature engineering, model validation, model tuning, model selection and deployment, machine learning interpretability, bring your own recipe, time-series and automatic pipeline generation for model scoring."

Cloud and OSS solution developers are less aggressive in claiming end-to-end support, since OSS could rely on programmatic interoperability with other libraries, and cloud providers in theory could integrate with their other offerings for other stages of the ML workflow. While intended for flexibility, the interoperable design led to a *fragmented data ecosystem*, causing users to *"spend most of the time gluing everything together."* (P14) In practice, all participants who used cloud-hosted Auto-ML reported using it in isolation, necessitating significant manual effort for data ingress and egress. P12, user of a cloud-based Auto-ML tool C lamented:

> "The biggest challenge is ... manipulating the data in a way that can be used with [C]. That is not something that we would have done if not for C. ... For each project you'll have to spend considerable amount of time structuring the data in a way that can be fed into C." (P12)

This drawback places cloud solutions below OSS in participant-rated completeness, despite the fact that cloud solutions provide more built-in features for model deployment and interpretability.

Limited data preprocessing. In terms of functionalities, a common complaint across all tools categories is inadequate support for data wrangling. As P1 pointed out, data preprocessing support in Auto-ML tools is primarily for feature engineering, which is deemed satisfactory by many Auto-ML users, and does not cover data cleaning and wrangling needs. In fact, P1, P3, and P15 stated that their Auto-ML tools did not support data preprocessing even though they acknowledged the feature engineering functionalities of their tools, since most of their time is spent on data wrangling and not feature engineering. P6's choice of Auto-ML tool was determined entirely by the tool's ability to automate domain specific data wrangling.

Among the host of data wrangling tasks enumerated in Section 6.3.2.1, participants expressed the need for improved system support for the following tasks:

- automated data discovery (P1)
- data cleaning for domain specific data (P6) and with more user control (P7) to avoid "garbage in, garbage out"
- data transformation for domain specific data (P9, P10, P13) and for mitigating common data problems such as class imbalance (P16)
- large-scale data processing using distributed architecture (P14)
- dataset augmentation using state-of-the-art research (P11).

Limited Support for complex models and data types. Classification and regression are currently the only types of tasks supported by Auto-ML tools. While some tools are beginning to support unstructured data such as text and images by harnessing recent development in deep learning, tabular data remains the focus for most Auto-ML tools. The existing user-base for Auto-ML is self selected to fit into the capabilities of current offerings. Even so, many participants expressed the desire for more broad-ranging support, such as unsupervised learning (P3, P6, P10, P15, P16) and domain-specific data models (P6: healthcare, P10: time series).

Causes system failures due to compute intensive workloads. A common complaint about Auto-ML by participants who used OSS solutions is system performance, a major contributing factor to OSS being rated lower than the other categories of tools on *ease of use* in Figure 6.1. P14 reported that *"running out of main memory was the biggest technical challenge."* P8, whose models had *18 million columns*, also reported that running out of memory, which crashed model training, was a frequent frustration. P5 had to run experiments on limited computation resources provided to her research lab, and she had to modify the model search space to reduce the total run time:

> "[It was] time consuming for [Auto-ML] for large dataset, and some pipelines are just too heavy and crash the process. For large feature set and sample set, some operators to further expand the data set were [slow and had to be] removed from the search space." (P5)

P11, P14, and P15 also reported that they needed to define the model search space carefully to cope with the compute-intensive nature of Auto-ML workloads, and P15 would even revert back to manual development for large models. The need to switch between development on laptops and on servers posed a major *"annoyance"* for P11. P5, P8, P11, P14, and P15 all reported spending only a fraction of their time on model development, thus their heavy Auto-ML compute needs are intermittent.

Lacks customizability. Customizability is the third lowest rated quality of Auto-ML tools by the participants, as shown in Figure 6.1. Interestingly, both too little customizability and too much customizability contributed to the low rating for customizability.

Wanting more customizability is a sentiment shared by many ML Engineers and ML Innovators, especially users of cloud-hosted solutions. Participants wanted more custom control for computation resource allocation (P2, P4), data cleaning procedures (P6, P7), model search space (P4, P13), and model interpretability techniques (P5).

On the other hand, too many customization options could lead to cognitive overload and hinder progress. P15 reported feeling overwhelmed by the number of hyperparameters that could be customized and needed to consult the documentation. P9, a novice, shared that they believed "there's a lot of tools higher than a five [for customizability] but in a bad way." P3 described the phenomenon of gratuitous customizability:

> "You don't necessarily know what some of the hyperparameters mean some of the time in extensive detail, but you do have the ability to control them all."

Most notably, P16 realized during the course of the interview that the additional customizability they wanted was in fact unnecessary for their use case:

> "I've kind of been harping on how it's not as customizable . . . But the tools that I've looked at lets you select the types of models to evaluate and change your features. They give you capability of managing the the information and shaping the underlying model . . . It's handled quite well."

We delve deeper into the issue of customizability and control in Section 6.3.5.

Lacks Transparency and Interpretability. The lowest rated quality of Auto-ML tools is transparency for the cloud solutions due to their black-box nature and relative lack of opportunities for user agency in comparison to other tool categories. For increased transparency and usability, a couple of participants expressed the desire for a simple progress bar that gives them insights into how long Auto-ML would take (P2, P13.) However, different user populations desire different levels of transparency to ensure trust:

> "Auto-ML is also an ML model. What that ML model is, how the ML model was trained, how the ML model learns from newer data—that piece is a black box. And so that makes it less trustworthy for people like me who are also ML engineers who knows the success probability of machine learning models. For someone who's not in ML, it's like magic. You just click a button and it works. But for someone who knows ML . . . I know one of the things that can go wrong [is] a ML model that is not trained well. One of the problems . . . with Auto-ML is that they don't give you a lot of information about what is actually going on behind the scenes, and that makes it really hard for me to trust." (P12)

Although widely reported by prior work on human-centered Auto-ML [53, 198] that transparency mechanisms increase user trust in Auto-ML systems, our results indicate that transparency mechanisms alone, such as visualization, do not suffice for the level of trust required in high-stake industrial settings, wherein participants need to reason and justify for how and why the model design and selection decisions are made. In order to gain the

level of interpretability and trust required for mission-critical projects, participants reported switching to complete manual development for increased user agency and control (P1, P4, P9, P11, P12, P15), as further discussed in Section 6.3.5. Conversely, lack of agency and tinkering can result in a lack of interpretability and the type of non-transparency caused by illiteracy [32]. Humans develop understanding by doing, as illustrated by P3:

> "You don't have a fundamental understanding of what's happening under the hood. And the other challenges with that ... are interpretability ... The onus is on me to actually build a competency in them ... It makes it basically impossible to go to a business person ... [to explain] how do you decide on this transaction. ... I actually have to go back ... and look at the 300 algorithms documentation, not the best one that I just deployed without really reading up all the details. There is a lot that you let go."

6.3.5 Roles of the Human in Auto-ML

Contrary to the moniker "automated ML", practitioners do not use Auto-ML tools as push-button, one-shot solutions. Instead, they collaborate with Auto-ML during the model development process—instructing, advising, and safeguarding Auto-ML. Humans are valuable contributors, mentors, and supervisors to Auto-ML, improving its efficiency, effectiveness, and safety. Their place "in the loop" cannot be replaced with complete automation. The consequences of removing humans out of the process are ineffectiveness, unpredictability, and harm to end users caused by spurious model behavior. Below we discuss the crucial roles humans play alongside Auto-ML.

6.3.5.1 Humans boost Auto-ML's performance and efficiency

Auto-ML, despite all its benefits, cannot be efficient and effective without humans in the loop, because humans' contextual awareness, domain expertise, and ML experience cannot be automated away. Humans are especially indispensable for non-standard uses cases and domains. Without human involvement, Auto-ML cannot meet the stringent requirements of real-world applications.

Human guidance constricts the search space of Auto-ML. The flip side of Auto-ML's comprehensiveness (as a benefit in Section 6.3.3) is the enormous search space that requires an unwieldy amount of compute and time resources, when attempting to maximize model performance (accuracy, or other performance metrics). The results in Chapter 3 and Chapter 4 show that a single model training can take hours. Due to Auto-ML's exhaustive search process, participants often only use Auto-ML for light and basic models that can be trained very quickly with a data set that is not too big (P9, P11, P14, P15). Auto-ML blindly searches through the entire space possible, resulting in intractable compute requirements, where "you will be there for years and years searching for models. It's not possible." (P11). Paradoxically, present day Auto-ML tools have no memory of and do not learn from the previous searches to narrow down the space in the next iteration, as P11 described:

> "But [Auto-ML] doesn't learn how to learn, and it doesn't learn the environment in which it learns ... I could put in more samples if I want to iterate again ... I have to do a whole new Auto-ML search, which is much more time consuming."

Humans, in contrast, learn from their previous experiences, and can guide Auto-ML using heuristics, thus narrowing the search space for Auto-ML, so that it can return outputs that meet the quality standards within time constraints. The most prevalent strategy that participants applied is to define the inclusion and/or exclusion criteria for models and/or hyperparameters (P7, P8, P12.) For example, P8 reported having to curate a list of models to feed into Auto-ML to constrain the search space. P14 also discussed the importance of manually limiting the hyperparameter search space:

> "I think one problem with Auto-ML is that it's very compute intensive. So you actually need to *define your space in a good way.* For example, if you want to find the range of hyperparameters, you need to have some kind of notion of what you want to use as initial parameters. So basically you have a trade-off between time and model accuracy." (P14)

P14's use case requires them to hold the time constraint as a constant, wherein model performance and search space size are inversely related, because for large search spaces untamed by human guidance, Auto-ML would 'cut corners' by skipping some search areas, weakening its performance. P9 described their mental model of Auto-ML's search process when they configured a time limit on Auto-ML runs:

> "It (Auto-ML) wouldn't necessarily do hyperparameter tuning. I'm only realistically going to be able to do that, if I'm using simpler models. Otherwise, I have to sit down with someone who knows the models really well and get a good default set of hyperparameters."

Humans compensate for Auto-ML shortcomings, boosting its performance. Auto-ML's shortcomings become more evident in non-standard use cases and domains (P4, P6, P9, P11, P12). In such cases, humans use Auto-ML 1) to establish an baseline for performance scores 2) to learn from Auto-ML's strategies 6.3.3 and manually improve ML model performance. For example, P4 reported:

> "When the task gets too complicated, Auto-ML breaks down ... you won't get good state-of-the-art results. As soon as the task becomes something out of the normal, I switch to manual. Auto-ML is just a good way to get the intuition behind what kind of performance you should expect in these cases. And then you try to beat that."

Auto-ML also often "over-thinks" and "over-complicates", resulting in diminished performance. Some participants respond by comparing manual and Auto-ML models and objectively select the best performing one based on certain performance metrics.

> "A couple of times, I ended up not using Auto-ML, because it was giving me a very complicated pipeline for regression problem. So I limited it to elastic net only, and that works better than what Auto-ML gave me. ... I select the best ones among manual and auto models."(P5)

Many participants describe their working relationship with Auto-ML as if they are collaborators [197], working in alternating cycles, iterating based on the feedback from each other (P10, P11, P13.) Humans leverage Auto-ML's strengths and compensate for its shortcomings, engaging with Auto-ML when it needs help. Together, they achieve higher performance (in speed and accuracy) than if they were each to work on their own.

For example, P10 jump-starts their ML workflow with a quick Auto-ML run to narrow down the variables/features, followed by manual feature engineering and refinement via several iterations with Auto-ML before finalizing the training sample for Auto-ML to build models with.

> "*What types of preprocessing is Auto-ML automating for you?* Oh, it's just sample selection. We can quickly run through modeling exercises and see which features or variables that we've tossed in are most important, and we're going to start chopping things out ... really quickly. And then go back to potentially doing some additional feature engineering ourselves manually or pulling more data in, but on a much limited scope ... And I'm going to continue to refine with multiple iterations of modeling, what ultimately I'm going to use as a training sample." (P10)

P10 compensates for Auto-ML's shortcoming in feature engineering, while also supporting Auto-ML with data collection—one of the tasks that are extremely to automate.

Humans do what Auto-ML cannot do at all, using contextual awareness and domain expertise. As reported in Section 6.2, many participants do not believe that tasks such as data preprocessing can ever be automated. The most common steps in the ML workflow where humans are indispensable are data collection (including data labeling), data cleaning, and feature engineering.(P1, P2, P3, P5, P6, P9, P10, P12, P14). Some ML tasks are extremely difficult to automate, such as unsupervised learning and semi-supervised learning (P7, P16.) Certain common data types also require substantial domain expertise, such as text (P9.) Automation's rigidity and lack of contextual awareness are well-studied in HCI [4]. The same is true for Auto-ML. As P6 puts it "Auto-ML is not smarter. It doesn't do what humans cannot do. It is just faster."

6.3.5.2 Humans increase ML safety and prevent misuse of Auto-ML

Participants reported Auto-ML's excessive ease of use to be a downfall (P9, P15). P10 suggested that Auto-ML is effective "under proper governance and management." P9 responded to the question "If an Auto-ML and a manual model have the same performance, in which model would you have more confidence?" with "It depends on the person who

used Auto-ML." To ensure the safety of Auto-ML decisions and prevent misuse, participants actively engage in the entire workflow as supervisors of Auto-ML, implementing governance and management. The common strategies include the following:

Humans compare manual and Auto-ML strategies as a safety check and to increase trust. Because of the shortcomings of Auto-ML, participants expressed the need to personally validate Auto-ML to establish confidence in Auto-ML models, which often entails manually developing models to compare with Auto-ML outputs (P5, P10, P11, P12, P16.) In addition, practitioners often engage with Auto-ML with a prior expectation of performance. As P16 reported:

> "There are certain times where running your own model, even if it's just as another perspective on the approach, and a **validation step** is still a good idea ... making sure that the direction of the predictions are in line with expectations [and] there's nothing abnormal happening."(P16)

Prior work found that ML interpretability seeks to establish trust not only between humans and models, but also between humans [81]. Participants reported comparing Auto-ML to manual ML for building trust between people.

> "My goal is to bring to my collaborators interpretable models ... the biggest challenge [with Auto-ML] is how to convince my collaborators to trust it. [The way I convince them is] we produce the standard approach models and show them that this one (Auto-ML) is actually better in performance terms." (P5)

Humans correct for Auto-ML idiosyncrasies. Ironically, as much as Auto-ML prevents suboptimal performance due to practitioners' idiosyncrasies (as mentioned in Section 6.3.3), humans correct for Auto-ML's idiosyncracies to safeguard Auto-ML outputs and improve overall performance. P10 describes one such issue with data leakage:

> "The only way that [Auto-ML]detects target leakages is if you have like a 99.99% correlation to the target label. So there's still things you need to know as a data scientists to use Auto-ML effectively."

P10 manually structured the data to ensure Auto-ML has the appropriate data as input to safeguard the model's generalizability and safety.

P16 worked with heavily imbalanced data and had to manually correct the biases in the data to prevent suboptimal performance before feeding the dataset to Auto-ML.

> "I know Auto-ML features allow you to do variable weighting. But ... I haven't found that to always work necessarily. "(P16)

Humans manually develop ML for understandability and reliability for mission-critical projects. Visibility into Auto-ML work process does not suffice for the level of understanding, trust, and explainability participants need for high-stake projects, where

humans are ultimately accountable for the reliable performance of ML models. Participants reported switching back to manually developing ML pipelines for mission-critical settings, because they need to reason and justify why the architecture and hyperparameters are chosen and how classification or prediction decisions are arrived at (P1, P4, P9, P11, P12, P15.)

Prior work concludes that transparency mechanisms, such as visualization, increase user understanding and trust in Auto-ML [53]. We found that transparency alone does not suffice for trust and understanding between humans and the tool-built model. Humans need agency to establish the level of understanding to trust Auto-ML. For example, P11, despite having visibility into Auto-ML, was afforded little understanding. P11 rated transparency of Auto-ML as being extremely transparent.

> "*To what extent do you have visibility into the inner workings of what the Auto-ML is doing?* If you want, it is a five (extremely visible). You can log whatever you want. And you can see what it's doing and it saves all the run and you can look at them in tensor board, you can even explore the architectures, so extremely [visible]."

However, P11 distrusted in Auto-ML and resorted to manual involvement.

> "[With] Auto-ML models, I can look at the architecture and I got no idea what it's doing. It is making connections all over the place. It is using non-conventional convolutional layers and you're like this is not predictable. Granted it's got good performance on one run ... You can't afford to have that kind of uncertainty ... You haven't had the **hands-on experience** to at least put your signature on the models and say put into production, so that's the issue."

P11 reported that lack of certainties and predictability in how Auto-ML selects model architecture undermined their trust in Auto-ML and that human agency increased model reliability and assures confidence in the model. This sentiment is also supported by other participants. For example P7, P13 prefer manual development for use cases where Auto-ML's strength in efficiency is irrelevant, because through manual ML development they can gain a deeper understanding and higher trust in the models.

6.4 Discussion

Our findings show that Auto-ML tools have been effective at making ML more accessible by creating high-level abstractions that streamline the process for training and deploying models. By effectively hiding away the complexities commonly associated with ML and acting as a source of knowledge, Auto-ML tools make ML accessible to novices and increase productivity for experts. However, we argue that current efforts in Auto-ML that strive for an eventual fully automated system through more tool capabilities and more accurate models fail to consider the ways the technology is actually used in practice and what users really need. In this work, we found that *complete automation is neither a requirement nor a desired*

outcome for users of Auto-ML. To make Auto-ML tools more effective, the goal should *not* be to completely remove the user from the process but instead to build human-compatible tools by working to achieve trust, understanding, and a sense of agency in the user. In this section, we discuss directions for the further development of Auto-ML tools based on our findings.

Adapt to the proficiency level of the intended user. Especially because Auto-ML tools mainly target citizen data scientists who often are intimated by ML due to perceived self-inefficacy, developers need to consider users' psychological readiness, designing from the position of a partner who has a sensitivity to the other's level of comfort (P12, P16.) P16 expressed this as follows:

> "A lot of the Auto-ML tools make an assumption about the technical competency of the user. That's to their detriment. I think if the goal is really to try and make it easier to use, there needs to be substantial effort put into the UX, and understanding of potential users that don't have... the depth of knowledge... [It is used] more as exploratory analytics or proof of concept."

Therefore Auto-ML tools need to be designed to adapt to users with varying comfort level instead of taking a one-size-fits-all approach.

Lingua Franca for ML. Developers of Auto-ML tools need to grapple with all the different roles that humans play in real-world data science work practices and how tools often need to take on a translator role among collaborators. P1 reported that even though many data science projects are initiated by business teams, the data scientists also often proactively propose innovative solutions to business teams based on their awareness of the challenges faced by business teams. ML engineers in real-world working environments do not simply passively react to requests from stakeholders. They also make contributions to the framing of the problem based on their ML expertise. Prior work highlights a translator role who sits between the data scientists and other stakeholders in collaborative data science projects [197]. We found that many ML engineers perform the role of the translator, directly interfacing with stakeholders, interpreting business problems and framing them into objectives that can be evaluated by ML. P9 described this challenge of mismatched mental models among collaborators:

> "The way business thinks about accuracy is often really really different from how you would calculate any sort of traditional metrics... You have to sync with a lot of people on exactly how they think accuracy works and how they want to report it."

Auto-ML tools need to be aware of this mismatch and adapt their language accordingly to proactively act as a translator. For example, data scientists measure model performance by accuracy whereas business analysts measure it by revenue generated.

Holistic platform. An important lesson to be learned from the large difference in favorability between the different categories of Auto-ML tools is that an *end-to-end* solution that

handles all stages of the ML workflow in a single environment is a highly desirable design choice for Auto-ML solutions. As mentioned in Section 6.1.1, Auto-ML platforms tend to have self-sufficient end-to-end ML workflow support due to limited options to integrate with external solutions, unlike OSS and Cloud Provider solutions. This has led to Auto-ML platforms being not only the most complete solution but also the easiest to use, the most efficient (no data transfer), and the most interpretable (comprehensive data lineage). Since Auto-ML is a highly complex system catered towards users with diverse backgrounds performing cognitively taxing tasks, interoperability with external solutions can add additional complexity. This is not to say that there is no use for non-platform Auto-ML solutions, but rather there needs to be a common substrate for the disparate Auto-ML tools and libraries, akin to Weld for data analytics [147]. Weld itself is too low level to serve as the desired substrate for Auto-ML, which needs to deal in model and task specifications.

Serverless Computing. As presented in Section 6.3.4, Auto-ML workloads are compute intensive but bursty, and the optimal hardware is highly variable depending on the dataset and model characteristics. These conditions motivate the need for elastic, ephemerally provisioned computation resources with variable specs. For example, for P8 whose dataset contained 18 million columns, they could be temporarily allocated machines with ample RAM, instead of the fixed-architecture university cluster they were using, to avoid crashing due to out-of-memory errors and accelerate model search. Recent advancements in serverless computing [17] can be harnessed to solve this problem. However, the Auto-ML setting poses new challenges in terms of the economics of trading compute cost for model accuracy.

Adaptive UI. Supporting users with diverse skills and expertise is an inherent challenge Auto-ML solutions must embrace. Evidence from Section 6.3.3 suggests that a blackbox interface for Auto-ML can be beneficial to novices and lower the maintenance overhead for all, but low customizability and transparency associated with blackbox interfaces hinder trust and agency based on evidence from Section 6.3.4. Auto-ML tool developers are forced to grapple with competing design objectives, and a natural solution to this conundrum is to provide multi-modal interfaces covering a spectrum of interaction levels.

The Auto-ML tooling landscape has progressed towards the low code/no code direction, with some solutions allowing experienced users to assume more control via a secondary programmatic interface. However, these tools do not offer true multi-modal interfaces but merely the option to export the raw model training code for further manual exploration, creating a clunky user interface "geared towards ML engineers who can learn to deal with broken UIs" (P12). Furthermore, the success of a multi-modal interface is contingent upon a user's ability to self select the most appropriate modality, which is not easily achievable as illustrated by P16's experience with customizability. A possible solution to this UX challenge is instead of having multiple distinct modalities, the Auto-ML tool can adaptively reveal or hide customization capabilities piecemeal, based on some approximation of user skill level and intent.

Interactive Exploration. Our findings in Section 6.3.5 show that a human in the loop can be an indispensable resource to complement Auto-ML and make it more efficient and

effective. In tools that support search space customization, human supervision is provided in a one-shot fashion. If the user wanted to iteratively refine the search space, they would have to manually kick off multiple rounds of the one-shot process, keeping track of intermediate results manually between iterations. Iterating with Auto-ML can be better supported by an interactive exploration interface designed specifically with iteration in mind. Such an interface needs to display summaries of all previous iterations and make it easy for the user to specify new search sub-spaces. It also needs to reconcile the high latencies of Auto-ML workloads and interactivity.

Balance between Human Control and Automation. In Section 6.3, we presented several deficiencies of the tools that can be improved with some of the suggestion above, as well as roles that are important for the human to assume when using Auto-ML. The decisions for what functionalities belong in either group are predicated upon, first and foremost, how to establish trust and agency in the human users, and the current state of ML, systems, and HCI research. Depriving users of trust and agency prevents Auto-ML from making an impact in real-world use cases, while attempting to automate certain features prematurely, without fundamental shifts in the underlying technology, requires strenuous efforts with little payoff in the user experience.

Certain tasks cannot be addressed with human control or automation alone but rather require a delicate balance between the two. Take data preprocessing for example. Evidence in Section 6.3.5 suggests that many users deem complex feature engineering simply out-of-reach for existing Auto-ML systems, due to their inability to capture domain knowledge. However, many also feel that existing tools lack basic support for mechanical tasks such as distributed processing and canonical data transformations (Section 6.3.4). Thus, the future of data preprocessing is a combination of adding support for mechanical tasks in the near term, and providing intuitive ways to specify high-level domain knowledge to integrate human intuition with Auto-ML long term.

Another example is the collaboration of humans and the machine for efficient hyperparameter tuning. While expert users believe that they can improve the efficiency of Auto-ML through intuition and experience, this belief is juxtaposed with their recognition that automated search can also correct for their biases and idiosyncrasies (Section 6.3.3). To strike a balance, we envision an interactive dialog between the human and the machine to iteratively discover and fill in each other's blindspots. Human guidance will be treated as strong priors on certain model subspaces but does not preclude the exploration of other subspaces, allowing the machine to nudge the human towards promising but underexplored subspaces.

6.5 Conclusion

Automation has become a central theme in ML tooling. This dissertation has proposed several ways to enhance ML tooling with intelligent automation. ML tools with automation has been given the moniker Auto-ML and have started to gain traction in recent year. In this chapter, we presented results from a semi-structured interview study of 16 practitioners

who have used Auto-ML in real-world applications to better understand what Auto-ML is today and what role automation should play in ML tooling. Our participants all reported various types of hybrid manual-auto strategies in leveraging Auto-ML in their development workflow, providing hints, safeguarding outputs, and massaging inputs into a form digestible by Auto-ML tools, among others. Current work practices around Auto-ML and perceptions of these tools demonstrate that complete automation of ML is neither realistic nor desirable. Our study sheds light on various forms of partnership or collaboration between humans and ML/AI, depending on user motivations, needs, skill-set, and use-cases, and provides next-generation Auto-ML tool developers with design guidelines for how to best incorporate pragmatic guidance and empower effective engagement with Auto-ML tools.

Chapter 7

Conclusion & Future Work

Machine learning has become a mainstream technology due to the previous decade's progress on programming interfaces for ML development and scalable data processing. Recent years have seen an explosion in the adoption of ML in a wide array of applications, creating new challenges around the usability and efficiency of systems for supporting ML, both in the development and deployment stages. In this dissertation, we conducted need-finding studies to better understand challenges in developing and deploying ML applications, thereby identifying opportunities for better tooling. We then presented solutions to address some of the opportunities uncovered by need finding, which we demonstrated to be effective due to the fact that they directly incorporate results from need finding to tackle the problem at hand.

In Chapter 2, our study results showed that ML development is driven by iterative trial-and-error, wherein the developer experiments with a large number of workflow configurations to achieve desired model performance. We observe that the configurations tend to have large overlaps that lead to redundant computation when executed in isolation, resulting in both computational resource inefficiencies and human inefficiencies. Developers are forced to stand idly by while the system churns through hours of redundant computation, disrupting continuity in their train of thought and lowering their productivity. To address this problem, Chapter 3 presented a solution that incorporates classic database techniques such as materialization and query rewriting to accelerate the development process by improving interactivity in data preprocessing and the end-to-end ML workflow development. Since the proposed solution was designed with findings from real-world workloads in mind, they were shown to be highly effective at tackling the original problems.

In Chapter 4, we found through a study of thousands of production ML pipelines that a vast amount of computation resources go into support ML applications in production, bringing resource utilization efficiency into focus. The corpus in this study contains highly complex provenance graphs of ML pipelines with continuous model updates, i.e., newly observed data automatically triggers the retrain and redeployment of the model. We proposed novel analysis techniques to work with the complexity of the provenance graph, which led to a wealth of interesting discoveries that present novel, fruitful directions for future research

on ML tooling. In Chapter 5, we focused specifically on the problem of improving resource utilization efficiency in the automatic retraining and redeployment of models in continuous pipelines. The ML-based solution we presented was shown to be able to eliminate over 50% of wasted computation without compromising the model refresh cadence by introducing intelligent execution policies to avoid pipeline runs that are not predicted to result in model refresh.

Finally, in Chapter 6, we turned our attention to the role of automation in ML development. As evident in the work presented in this dissertation as well as general research trends, automation is playing an increasingly prominent role in the ML tooling landscape, giving rise to the new field of Auto-ML. Efforts to make machine learning more widely accessible have led to a rapid increase in Auto-ML tools that aim to automate the process of training and deploying machine learning. To understand how Auto-ML tools are used in practice today, we performed a qualitative study with participants ranging from novice hobbyists to industry researchers who use Auto-ML tools. We presented insights into the benefits and deficiencies of existing tools, as well as the respective roles of the human and automation in ML workflows. We discussed design implications for the future of Auto-ML tool development and argued that instead of full automation being the ultimate goal of Auto-ML, designers of these tools should focus on supporting a partnership between the user and the Auto-ML tool.

Our work in this dissertation is part of a broader agenda towards the democratization of machine learning and data science. Standing between ML and those who wish to adopt ML are the tools for building and deploying ML workflows. Thus, the design and implementation of such tools have profound impact on the accessibility of ML to a wide range of audience. This dissertation aims to increase accessibility by designing solutions to lower the skill and resource barriers for ML adoption, based on insights gleaned from studies of existing user behavior and system performance. Our efforts are simply a small step towards the broader agenda. We provide a few examples of fruitful future directions below.

7.1 Future Work

Understanding User Behavior. While our studies in Chapter 2 surfaced many interesting patterns in iterative ML workflow development, our datasets provide a limited view into the user's thought process behind performing the manual iterations, as shown in the case studies. A more in-depth user study is a promising direction for future work towards understanding the motivations and cognitive processes behind ML model development. This understanding can form the basis of a more effective and efficient Auto-ML strategy by mimicking human experts. Likewise, human-in-the-loop ML systems can benefit from automated guidance suggesting areas of exploration that the ML developer may not have thought of. Further work along this line will shed light on design insights for building more usable and more intelligent machine learning development systems.

Supporting ML Development. The solution we proposed for ML development focused on accelerating the execution of user-designed experiments. However, our need finding studies in Chapter 2 suggest that users often are aimlessly experimenting with random configurations in hopes of happening upon an effective workflow. In such scenarios, the system can make intelligent recommendations on new experiments to bypass the need for exhaustive, unfocused search. Furthermore, the work in this dissertation focused primarily on supporting a linear sequence of human-in-the-loop experiments for developing ML workflows. As ML tooling trends more towards automation, the experimentation process is becoming increasingly parallel, posing new challenges in data management across concurrently evaluated ML workflows that share intermediate results.

Supporting ML Deployment. Chapter 4 discussed many research opportunities exposed by the large-scale study of production ML pipelines, including the fact that 1) vocabulary computation over categorical features are highly prevalent in production ML and can benefit a great deal from data management techniques for improved efficiency, 2) a wide variety of model types are run in production at the same organization, motivating the need for a common platform that is able to support different types of learning, 3) consecutive runs of the same pipeline for model update consume overlapping data spans, leading to redundant computation that can be eliminated through better data management, 4) many pipeline runs fail to produce an updated model, leading to wasted computation that has no effect on downstream applications. We presented a solution for the last problem in this dissertation, but the other opportunities are equally worth pursuing.

Human-ML Collaboration. At the end of Chapter 6, we discussed several ways to design and implement tools towards better human-ML collaboration. We argued that the viable future of ML tooling is not removing the human from the loop but rather augmenting the human via better human-ML collaboration, which can come in many different flavors depending on the use case and user persona. For example, a novice user might want more guidance in the experimentation process with recommendations generated from best practices and potentially tailor-fitted to their specific use cases. On the other hand, expert ML practitioners might be looking for capabilities to automate repetitive data management tasks. In either case, the tool must not take away agency from the user, and the tool behavior must be explainable and audible. While we offered some core insights into the principles behind designing for better human-ML collaboration in Chapter 6, acting on these principles to implement usable tools remains an open problem.

Bibliography

[1] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Ashraf Abdul et al. "Trends and Trajectories for Explainable, Accountable and Intelligible Systems: An HCI Research Agenda". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. 2018, p. 582.

[3] Rui Abreu et al. "Provenance Segmentation". In: *8th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2016, Washington, D.C., USA, June 8-9, 2016*. Ed. by Sarah Cohen Boulakia. USENIX Association, 2016. URL: https://www.usenix.org/conference/tapp16/workshop-program/presentation/abreu.

[4] Mark S. Ackerman. "The Intellectual Challenge of CSCW: The Gap between Social Requirements and Technical Feasibility". In: *Hum.-Comput. Interact.* 15.2 (Sept. 2000), pp. 179–203. ISSN: 0737-0024. DOI: 10.1207/S15327051HCI1523_5. URL: https://doi.org/10.1207/S15327051HCI1523_5.

[5] Deepak Agarwal et al. "Laser: A scalable response prediction platform for online advertising". In: *Proceedings of the 7th ACM international conference on Web search and data mining*. 2014, pp. 173–182.

[6] Google AI. *Facets: An Open Source Visualization Tool for Machine Learning Training Data*. 2017. URL: https://ai.googleblog.com/2017/07/facets-open-source-visualization-tool.html.

[7] Eleanor Ainy et al. "Approximated Summarization of Data Provenance". In: *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*. Ed. by James Bailey et al. ACM, 2015, pp. 483–492. DOI: 10.1145/2806416.2806429. URL: https://doi.org/10.1145/2806416.2806429.

[8] AmazonSagemakerAutopilot. *Automate model development with Amazon SageMaker Autopilot*. Website. 2020. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/autopilot-automate-model-development.html.

[9] Saleema Amershi et al. "ModelTracker: Redesigning Performance Analysis Tools for Machine Learning". In: *Proceedings of the 33rd ACM SIGCHI Conference on Human Factors in Computing Systems*. Seoul, Korea: ACM, 2015, pp. 337–346. ISBN: 978-1-4503-3145-6. DOI: 10.1145/2702123.2702509.

[10] Saleema Amershi et al. "Power to the People: The Role of Humans in Interactive Machine Learning". In: *AI Magazine* 35.4 (2014), pp. 105–120. ISSN: 0738-4602. DOI: 10.1609/aimag.v35i4.2513. URL: http://aaaipress.org/ojs/index.php/aimagazine/article/view/2513.

[11] Saleema Amershi et al. "Software Engineering for Machine Learning: A Case Study". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042. URL: https://doi.org/10.1109/ICSE-SEIP.2019.00042.

[12] Yael Amsterdamer et al. "Putting Lipstick on Pig: Enabling Database-style Workflow Provenance". In: *Proc. VLDB Endow.* 5.4 (2011), pp. 346–357. DOI: 10.14778/2095686.2095693. URL: http://vldb.org/pvldb/vol5/p346%5C_yaelamsterdamer%5C_vldb2012.pdf.

[13] Michael R Anderson and Michael Cafarella. "Input selection for fast feature engineering". In: *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016, pp. 577–588.

[14] Michael R Anderson et al. "Brainwash: A Data System for Feature Engineering." In: *CIDR*. 2013.

[15] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[16] Alexander Ratner Stephen H Bach et al. "Snorkel: Rapid Training Data Creation with Weak Supervision". In: *Proceedings of the VLDB Endowment* 11.3 (2017), pp. 269–282.

[17] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research Advances in Cloud Computing*. none: Springer, 2017, pp. 1–20.

[18] Zhuowei Bao et al. "An optimal labeling scheme for workflow provenance using skeleton labels". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 711–722. DOI: 10.1145/1807167.1807244. URL: https://doi.org/10.1145/1807167.1807244.

[19] Jeff Barnes. *Azure machine learning microsoft azure essentials*. 2015.

[20] Louis Bavoil et al. "VisTrails: Enabling Interactive Multiple-View Visualizations". In: *16th IEEE Visualization Conference, IEEE Vis 2005, Minneapolis, MN, USA, October 23-28, 2005, Proceedings*. IEEE Computer Society, 2005, pp. 135–142. DOI: 10.1109/VISUAL.2005.1532788. URL: https://doi.org/10.1109/VISUAL.2005.1532788.

[21] Denis Baylor et al. "Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform". In: *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*. Ed. by Bharath Ramsundar and Nisha Talagala. USENIX Association, 2019, pp. 51–53. URL: https://www.usenix.org/conference/opml19/presentation/baylor.

[22] Denis Baylor et al. "Tfx: A tensorflow-based production-scale machine learning platform". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2017, pp. 1387–1395.

[23] Maxime Beauchemin. *Airflow: a workflow management platform*. https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8. Accessed October 8, 2018.

[24] Besim Bilalli, Alberto Abello, and Tomas Aluja-Banet. "On the Predictive Power of Meta-Features in OpenML". In: *Int. J. Appl. Math. Comput. Sci* 27.4 (2017), pp. 697–712. DOI: 10.1515/amcs-2017-0048.

[25] Besim Bilalli et al. "PRESISTANT: Learning based assistant for data pre-processing". In: *Data and Knowledge Engineering* (2019). ISSN: 0169023X. DOI: 10.1016/j.datak.2019.101727. arXiv: 1803.01024.

[26] Olivier Biton et al. "Querying and Managing Provenance through User Views in Scientific Workflows". In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE Computer Society, 2008, pp. 1072–1081. DOI: 10.1109/ICDE.2008.4497516. URL: https://doi.org/10.1109/ICDE.2008.4497516.

[27] Matthias Boehm et al. "Declarative Machine Learning-A Classification of Basic Properties and Types". In: *arXiv preprint arXiv:1605.05826* (2016).

[28] Eric Breck et al. "Data Validation for Machine Learning". In: *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. Ed. by Ameet Talwalkar, Virginia Smith, and Matei Zaharia. mlsys.org, 2019. URL: https://proceedings.mlsys.org/book/267.pdf.

[29] Eran Bringer et al. "Osprey: Weak Supervision of Imbalanced Extraction Problems without Code". In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*. DEEM'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450367974. DOI: 10.1145/3329486.3329492. URL: https://doi.org/10.1145/3329486.3329492.

[30] Ivan Bruha and A Famili. "Postprocessing in machine learning and data mining". In: vol. 2. 2. ACM, 2000, pp. 110–114.

[31] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[32] Jenna Burrell. "How the machine 'thinks': Understanding opacity in machine learning algorithms". In: *Big Data & Society* 3.1 (2016), p. 2053951715622512. DOI: 10.1177/2053951715622512. eprint: https://doi.org/10.1177/2053951715622512. URL: https://doi.org/10.1177/2053951715622512.

[33] Steven P Callahan et al. "VisTrails: visualization meets data management". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 745–747.

[34] William La Cava et al. *Evaluating Recommender System for AI-Driven Data Science: A Preprint*. Tech. rep. 2019. arXiv: 1905.09205v2. URL: http://automl.chalearn.org/.

[35] Eugen Cepoi and Liping Peng. "Runway - Model Lifecycle Management at Netflix". In: USENIX Association, July 2020.

[36] Lingjiao Chen et al. "Towards linear algebra over normalized data". In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1214–1225.

[37] James Cheney, Laura Chiticariu, and Wang Chiew Tan. "Provenance in Databases: Why, How, and Where". In: *Found. Trends Databases* 1.4 (2009), pp. 379–474. DOI: 10.1561/1900000006. URL: https://doi.org/10.1561/1900000006.

[38] Rada Chirkova, Jun Yang, et al. "Materialized views". In: *Foundations and Trends® in Databases* 4.4 (2012), pp. 295–405.

[39] Corinna Cortes et al. *AdaNet: Adaptive Structural Learning of Artificial Neural Networks*. 2017. arXiv: 1607.01097 [cs.LG].

[40] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[41] Andrew Crotty et al. "An architecture for compiling udf-centric workflows". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1466–1477.

[42] Aymeric Damien et al. *Tflearn*. 2016.

[43] *Data Driven Discovery of Models*. URL: https://docs.datadrivendiscovery.org/.

[44] Datarobot. *DataRobot Automated Machine Learning*. Website. 2020. URL: https://www.datarobot.com/platform/automated-machine-learning/.

[45] Christopher De Sa et al. "DeepDive: Declarative Knowledge Base Construction". In: *SIGMOD Rec.* 45.1 (June 2016), pp. 60–67. ISSN: 0163-5808. DOI: 10.1145/2949741.2949756. URL: http://doi.acm.org/10.1145/2949741.2949756.

[46] dedoose. *Dedoose*. Website. 2020. URL: https://www.dedoose.com.

[47] Ewa Deelman et al. "Pegasus: Mapping scientific workflows onto the grid". In: *Grid Computing*. Springer. 2004, pp. 11–20.

[48] *DeepDive Census example*. https://github.com/HazyResearch/deepdive/tree/master/examples/census.

[49] *Deeplearning4j: Open-source distributed deep learning for the JVM*.

[50] Dua Dheeru and Efi Karra Taniskidou. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[51] Pedro Domingos. "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10 (2012), pp. 78–87.

[52] Mike Dreves et al. "From Data to Models and Back". In: *Proceedings of the Fourth Workshop on Data Management for End-To-End Machine Learning, In conjunction with the 2020 ACM SIGMOD/PODS Conference, DEEM@SIGMOD 2020, Portland, OR, USA, June 14, 2020*. Ed. by Sebastian Schelter, Steven Whang, and Julia Stoyanovich. ACM, 2020, 1:1–1:4. DOI: 10.1145/3399579.3399868. URL: https://doi.org/10.1145/3399579.3399868.

[53] Jaimie Drozdal et al. "Trust in AutoML: Exploring Information Needs for Establishing Trust in Automated Machine Learning Systems". In: *Proceedings of the 25th International Conference on Intelligent User Interfaces*. IUI '20. Cagliari, Italy: Association for Computing Machinery, 2020, pp. 297–307. ISBN: 9781450371186. DOI: 10.1145/3377325.3377501. URL: https://doi.org/10.1145/3377325.3377501.

[54] Jeffrey Dunn. *Introducing FBLearner Flow: Facebook's AI backbone*. https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/. Accessed October 8, 2018.

[55] Sylvain Duranton et al. *What's Keeping Women Out of Data Science?* https://www.bcg.com/en-us/publications/2020/what-keeps-women-out-data-science. 2020.

[56] Jack Edmonds and Richard M Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[57] Iman Elghandour and Ashraf Aboulnaga. "ReStore: reusing results of MapReduce jobs". In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 586–597.

[58] Jerry Alan Fails and Dan R. Olsen. "Interactive Machine Learning". In: *Proceedings of the 8th International Conference on Intelligent User Interfaces*. IUI '03. Miami, Florida, USA: Association for Computing Machinery, 2003, pp. 39–45. ISBN: 1581135866. DOI: 10.1145/604045.604056. URL: https://doi.org/10.1145/604045.604056.

[59] Xixuan Feng et al. "Towards a unified architecture for in-RDBMS analytics". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 325–336.

[60] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2755–2763.

[61] Rebecca Fiebrink, Perry R Cook, and Daniel Trueman. "Human Model Evaluation in Interactive Supervised Learning". In: *CHI 2011* 4.2 (2011), pp. 147–156. ISSN: 23138734. DOI: 10.14529/jsfi170202. URL: http://superfri.org/superfri/article/view/133.

[62] Juliana Freire et al. "Provenance for Computational Tasks: A Survey". In: *Comput. Sci. Eng.* 10.3 (2008), pp. 11–21. DOI: 10.1109/MCSE.2008.79. URL: https://doi.org/10.1109/MCSE.2008.79.

[63] Nicolo Fusi, Rishit Sheth, and Melih Huseyn Elibol. "Probabilistic Matrix Factorization for Automated Machine Learning". In: NeurIPS, 2018. arXiv: 1705.05355v2.

[64] João Gama et al. "A survey on concept drift adaptation". In: *ACM computing surveys (CSUR)* 46.4 (2014), pp. 1–37.

[65] Amol Ghoting et al. "SystemML: Declarative machine learning on MapReduce". In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 231–242.

[66] Pieter Gijsbers et al. "An Open Source AutoML Benchmark". In: (July 2019). arXiv: 1907.00909. URL: http://arxiv.org/abs/1907.00909.

[67] Yolanda Gil et al. "Towards Human-Guided Machine Learning". In: *Proceedings of the 24th International Conference on Intelligent User Interfaces*. IUI '19. Marina del Ray, California: Association for Computing Machinery, 2019, pp. 614–624. ISBN: 9781450362726. DOI: 10.1145/3301275.3302324. URL: https://doi.org/10.1145/3301275.3302324.

[68] GoogleCloudAutoML. *Google Cloud AutoML*. Website. 2020. URL: https://cloud.google.com/automl.

[69] Andrew D Gordon. "A tutorial on co-induction and functional programming". In: *Functional Programming, Glasgow 1994*. Springer, 1995, pp. 78–95.

[70] Pradeep Kumar Gunda et al. "Nectar: Automatic Management of Data and Computation in Datacenters." In: *OSDI*. Vol. 10. 2010, pp. 1–8.

[71] H2O.ai. *H2o.ai Automated Machine Learning*. Website. 2020. URL: https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html#automl-interface.

[72] Alon Y. Halevy et al. "Goods: Organizing Google's Datasets". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 795–806. DOI: 10.1145/2882903.2903730. URL: https://doi.org/10.1145/2882903.2903730.

[73] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[74] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *CIDR*. 2017.

[75] Joseph M Hellerstein et al. "The MADlib analytics library: or MAD skills, the SQL". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1700–1711.

[76] Joseph M. Hellerstein et al. "The MADlib Analytics Library or MAD Skills, the SQL". In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1700–1711. DOI: 10.14778/2367502.2367510. URL: http://vldb.org/pvldb/vol5/p1700%5C_joehellerstein%5C_vldb2012.pdf.

[77] Dorit S Hochbaum and Anna Chen. "Performance analysis and best implementations of old and new algorithms for the open-pit mining problem". In: *Operations Research* 48.6 (2000), pp. 894–914.

[78] Fred Hohman et al. "Understanding and Visualizing Data Iteration in Machine Learning". In: *CHI '20: Proceedings of the 2020 annual conference on Human factors in computing systems* (2020), pp. 1–13.

[79] Fred Hohman et al. "Understanding and Visualizing Data Iteration in Machine Learning". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376177. URL: https://doi.org/10.1145/3313831.3376177.

[80] David A Holland et al. "Choosing a data model and query language for provenance". In: *Proceedings of the 2nd International Provenance and Annotation Workshop (IPAW'08)*. Springer. 2008.

[81] Sungsoo Ray Hong, Jessica Hullman, and Enrico Bertini. "Human Factors in Model Interpretability: Industry Practices, Challenges, and Needs". In: *Proceedings of the ACM on Human-Computer Interaction* 4.CSCW1 (2020), pp. 1–26.

[82] Matteo Interlandi et al. "Titian: Data Provenance Support in Spark". In: *Proc. VLDB Endow.* 9.3 (2015), pp. 216–227. DOI: 10.14778/2850583.2850595. URL: http://www.vldb.org/pvldb/vol9/p216-interlandi.pdf.

[83] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 1946–1956. ISBN: 9781450362016. DOI: 10.1145/3292500.3330648. URL: https://doi.org/10.1145/3292500.3330648.

[84] Alekh Jindal et al. "Selecting subexpressions to materialize at datacenter scale". In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 800–812.

[85] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349 (2015), pp. 255–260. URL: http://science.sciencemag.org/content/349/6245/255.

[86] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[87] S. Kandel et al. "Enterprise Data Analysis and Visualization: An Interview Study". In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (Dec. 2012), pp. 2917–2926. ISSN: 2160-9306. DOI: 10.1109/TVCG.2012.219.

[88] Sean Kandel et al. "Enterprise data analysis and visualization: An interview study". In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926.

[89] Konstantinos Karanasos et al. "Extending relational query processing with ML inference". In: *arXiv preprint arXiv:1911.00231* (2019).

[90] Andrej Karpathy and Li Fei-Fei. "Deep visual-semantic alignments for generating image descriptions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3128–3137.

[91] Konstantinos Katsiapis and Kevin Haas. "Towards ML Engineering with TensorFlow Extended (TFX)". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. Ed. by Ankur Teredesai et al. ACM, 2019, p. 3182. DOI: 10.1145/3292500.3340408. URL: https://doi.org/10.1145/3292500.3340408.

[92] Shachar Kaufman et al. "Leakage in data mining: Formulation, detection, and avoidance". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6.4 (2012), pp. 1–21.

[93] Mary Beth Kery, Amber Horvath, and Brad A Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In: *CHI*. 2017, pp. 1265–1276.

[94] John King and Roger Magoulas. *Data science salary survey: tools, trends, what pays (and what doesn't) for data professionals*. 2016.

[95] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education, 2006.

[96] Christoph Koch, Daniel Lupei, and Val Tannen. "Incremental View Maintenance For Collection Programming". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 75–90. ISBN: 9781450341912. DOI: 10.1145/2902251.2902286. URL: https://doi.org/10.1145/2902251.2902286.

[97] Laura M Koesten et al. "The Trials and Tribulations of Working with Structured Data:-a Study on Information Seeking Behaviour". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. 2017, pp. 1277–1289.

[98] Ron Kohavi. "Scaling up the accuracy of Naive-Bayes classifiers: a decision-tree hybrid". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press. 1996, pp. 202–207.

[99] Sanjay Krishnan et al. "Activeclean: Interactive data cleaning while learning convex loss models". In: *arXiv preprint arXiv:1601.03797* (2016).

[100] *Kubeflow*. accessed 2020-09.

[101] Arun Kumar. "CEREBRO: A System to Manage Deep Learning for Relational Data Analytics." In: *CIDR*. 2017.

[102] Arun Kumar, Matthias Boehm, and Jun Yang. "Data management in machine learning: Challenges, techniques, and systems". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1717–1722.

[103] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. "Learning generalized linear models over normalized data". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1969–1984.

[104] Arun Kumar et al. "Model Selection Management Systems: The Next Frontier of Advanced Analytics". In: *SIGMOD Rec.* 44.4 (2015), pp. 17–22. DOI: 10.1145/2935694.2935698. URL: https://doi.org/10.1145/2935694.2935698.

[105] Arun Kumar et al. "To join or not to join?: Thinking twice about joins before feature selection". In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 19–34.

[106] John Langford, Lihong Li, and Alex Strehl. *Vowpal wabbit online learning project*. 2007.

[107] Trang T Le, Weixuan Fu, and Jason H Moore. "Scaling tree-based automated machine learning to biomedical big data with a feature set selector". In: *Bioinformatics* 36.1 (2020), pp. 250–256.

[108] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[109] Angela Lee et al. "Demystifying a Dark Art: Understanding Real-World Machine Learning Model Development". In: *Workshop on Human-in-the-Loop Data Analytics (HILDA) at the ACM SIGMOD International Conference on Management of Data* (2020).

[110] Doris Jung-Lin Lee et al. "A Human-in-the-loop Perspective on AutoML: Milestones and the Road Ahead". In: *Data Engineering* 58 (2019).

[111] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: http://jmlr.org/papers/v18/16-365.

[112] Mu Li et al. "Scaling distributed machine learning with the parameter server". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[113] Edo Liberty et al. "Elastic Machine Learning Algorithms in Amazon SageMaker". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 731–737. ISBN: 9781450367356. DOI: 10.1145/3318464.3386126. URL: https://doi.org/10.1145/3318464.3386126.

[114] Yong Liu et al. "CHI 1994-2013: mapping two decades of intellectual progress through co-word analysis". In: *proceedings of the SIGCHI conference on human factors in computing systems*. ACM. 2014, pp. 3553–3562.

[115] Yucheng Low et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[116] Bertram Ludäscher et al. "Scientific workflow management and the Kepler system". In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[117] *Machine Learning Library (MLlib) Guide*. accessed 2020-11.

[118] *Machine Learning: The New Proving Ground for Competitive Advantage*. 2017. URL: https://s3.amazonaws.com/files.technologyreview.com/whitepapers/MITTR_GoogleforWork_Survey.pdf.

[119] "Machine learning: the power and promise of computers that learn by example". In: (2017). URL: https://royalsociety.org/~/media/policy/projects/machine-learning/publications/machine-learning-report.pdf.

[120] Christopher D Manning et al. "The Stanford CoreNLP Natural Language Processing Toolkit." In: *ACL (System Demonstrations)*. 2014, pp. 55–60.

[121] Xian-Ling Mao et al. "S2JSD-LSH: A locality-sensitive hashing schema for probability distributions". In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[122] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. "Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice". In: *Proceedings of the ACM on Human-Computer Interaction* 3 (Nov. 2019), pp. 1–23. DOI: 10.1145/3359174.

[123] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX: SciPy 2010, 2010, pp. 51–56.

[124] *Meet Michelangelo: Uber's Machine Learning Platform*. https://eng.uber.com/michelangelo/. Accessed October 8, 2018.

[125] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 706–706. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142552. URL: http://doi.acm.org/10.1145/1142473.1142552.

[126] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: (2016).

[127] *Metaflow*. accessed 2020-09.

[128] Hui Miao and Amol Deshpande. "ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows". In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 26–38. URL: http://sites.computer.org/debull/A18dec/p26.pdf.

[129] Hui Miao and Amol Deshpande. "Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 1710–1713. DOI: 10.1109/ICDE.2019.00179. URL: https://doi.org/10.1109/ICDE.2019.00179.

[130] MicrosoftAzureAutomatedML. *Microsoft Azure Automated Machine Learning*. Website. 2020. URL: https://azure.microsoft.com/en-us/services/machine-learning/automatedml/.

[131] Tomas Mikolov et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[132] *ML Metadata*. accessed 2020-09.

[133] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. *Ludwig: a type-based declarative deep learning toolbox*. 2019. arXiv: 1909.07930 [cs.LG].

[134] Luc Moreau. "Aggregation by Provenance Types: A Technique for Summarising Provenance Graphs". In: *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015*. Ed. by Arend Rensink and Eduardo Zambon. Vol. 181. EPTCS. 2015, pp. 129–144. DOI: 10.4204/EPTCS.181.9. URL: https://doi.org/10.4204/EPTCS.181.9.

[135] Luc Moreau et al. *Prov-dm: The prov data model*. 2013.

[136] Luc Moreau et al. "The Open Provenance Model core specification (v1.1)". In: *Future Gener. Comput. Syst.* 27.6 (2011), pp. 743–756. DOI: 10.1016/j.future.2010.07.005. URL: https://doi.org/10.1016/j.future.2010.07.005.

[137] M Arthur Munson. "A study on the importance of and time spent on different modeling steps". In: *ACM SIGKDD Explorations Newsletter* 13.2 (2012), pp. 65–71.

[138] Nazri Mohd Nawi, Walid Hasen Atomi, and Mohammad Zubair Rehman. "The effect of data pre-processing on optimized training of artificial neural networks". In: *Procedia Technology* 11 (2013), pp. 32–39.

[139] Anton J Nederhof. "Methods of coping with social desirability bias: A review". In: *European journal of social psychology* 15.3 (1985), pp. 263–280.

[140] Randal S. Olson et al. "PMLB: A large benchmark suite for machine learning evaluation and comparison". In: *BioData Mining* 10.1 (Dec. 2017), p. 36. ISSN: 17560381. DOI: 10.1186/s13040-017-0154-4. arXiv: 1703.00512. URL: https://biodatamining.biomedcentral.com/articles/10.1186/s13040-017-0154-4.

[141] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[142] Dan Olteanu. "The Relational Data Borg is Learning". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3502–3515. URL: http://www.vldb.org/pvldb/vol13/p3502-olteanu.pdf.

[143] Dan Olteanu and Maximilian Schleich. "F: regression models over factorized views". In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1573–1576.

[144] Jorge Piazentin Ono et al. *PipelineProfiler: A Visual Analytics Tool for the Exploration of AutoML Pipelines*. 2020. arXiv: 2005.00160 [cs.HC].

[145] Sean Owen et al. "Mahout in action". In: (2012).

[146] *Pachyderm*. accessed 2020-09.

[147] Shoumik Palkar et al. "Weld: A common runtime for high performance data analytics". In: *Conference on Innovative Data Systems Research (CIDR)*. Chaminade, California: CIDR, 2017, p. 45.

[148] Adam Paszke et al. *Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration*. 2017.

[149] Kayur Patel et al. "Gestalt: integrated support for implementation and analysis in machine learning". In: *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM. 2010, pp. 37–46.

[150] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[151] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12.10 (2011), pp. 2825–2830.

[152] Luis L Perez and Christopher M Jermaine. "History-aware query optimization with materialized intermediate views". In: *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE. 2014, pp. 520–531.

[153] João Felipe Pimentel et al. "noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1841–1844. DOI: 10.14778/3137765.3137789. URL: http://www.vldb.org/pvldb/vol10/p1841-pimentel.pdf.

[154] Andrew M Pitts. "Operationally-based theories of program equivalence". In: *Semantics and Logics of Computation* 14 (1997), p. 241.

[155] Neoklis Polyzotis et al. "Data lifecycle challenges in production machine learning: a survey". In: *ACM SIGMOD Record* 47.2 (2018), pp. 17–28.

[156] Neoklis Polyzotis et al. "Data management challenges in production machine learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1723–1726.

[157] Martijn J. Post, Peter Van Der Putten, and Jan N. Van Rijn. "Does feature selection improve classification? A large scale experiment in OpenML". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9897 LNCS. Springer Verlag, 2016, pp. 158–170. ISBN: 9783319463483. DOI: 10.1007/978-3-319-46349-0_14.

[158] Fotis Psallidas et al. "Data Science through the looking glass and what we found there". In: (2019). arXiv: 1912.09536. URL: http://arxiv.org/abs/1912.09536.

[159] Junfei Qiu et al. "A survey of machine learning for big data processing". In: *EURASIP Journal on Advances in Signal Processing* 2016.1 (2016), p. 67.

[160] Gonzalo Ramos et al. "Interactive machine teaching: a human-centered approach to building machine-learned models". In: *Human-Computer Interaction* 35 (Apr. 2020), pp. 1–39. DOI: 10.1080/07370024.2020.1734931.

[161] WS Rasband. "ImageJ: Image processing and analysis in Java". In: *Astrophysics Source Code Library* (2012).

[162] Alexander Ratner et al. "Snorkel: Rapid training data creation with weak supervision". In: *arXiv preprint arXiv:1711.10160* (2017).

[163] Benjamin Recht et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems*. 2011, pp. 693–701.

[164] Xiang Ren et al. "Life-iNet: A Structured Network-Based Knowledge Exploration and Analytics System for Life Sciences". In: *Proceedings of ACL 2017, System Demonstrations* (2017), pp. 55–60.

[165] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[166] Yuji Roh, Geon Heo, and Steven Euijong Whang. *A Survey on Data Collection for Machine Learning: a Big Data – AI Integration Perspective*. 2019. arXiv: 1811.03402 [cs.LG].

[167] Joshua Rosen. *PySpark Internals*.

[168] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. "The earth mover's distance as a metric for image retrieval". In: *International journal of computer vision* 40.2 (2000), pp. 99–121.

[169] Lukas Rupprecht et al. "Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3354–3368. URL: http://www.vldb.org/pvldb/vol13/p3354-rupprecht.pdf.

[170] Carlton E. Sapp. "Preparing and Architecting for Machine Learning". In: (2017).

[171] Sebastian Schelter et al. "Automatically tracking metadata and provenance of machine learning experiments". In: *Machine Learning Systems workshop at NIPS*. 2017.

[172] Sebastian Schelter et al. "On Challenges in Machine Learning Model Management." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 5–15.

[173] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[174] *Scikit-Learn User Guide*. http://scikit-learn.org/stable/user_guide.html.

[175] David Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems* 28 (2015), pp. 2503–2511.

[176] *sklearn.pipeline.Pipeline*. accessed 2020-11.

[177] Evan R Sparks et al. "Keystoneml: Optimizing pipelines for large-scale advanced analytics". In: *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE. 2017, pp. 535–546.

[178] Vikram Sreekanti et al. "Optimizing Prediction Serving on Low-Latency Serverless Dataflow". In: *arXiv preprint arXiv:2007.05832* (2020).

[179] Michael Stonebraker et al. "The architecture of SciDB". In: *International Conference on Scientific and Statistical Database Management*. Springer. 2011, pp. 1–16.

[180] Benjamin Strang et al. "Don't rule out simple models prematurely: A large scale benchmark comparing linear and non-linear classifiers in OpenML". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11191 LNCS. Springer Verlag, 2018, pp. 303–315. ISBN: 9783030017675. DOI: 10.1007/978-3-030-01768-2_25.

[181] Arvind Sujeeth et al. "OptiML: an implicitly parallel domain-specific language for machine learning". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 609–616.

[182] Roshan Sumbaly, Jay Kreps, and Sam Shah. "The big data ecosystem at linkedin". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 1125–1134.

[183] Justin Talbot et al. "EnsembleMatrix: Interactive Visualization to Support Machine Learning with Multiple Classifiers". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 1283–1292. ISBN: 9781605582467. DOI: 10.1145/1518701.1518895. URL: https://doi.org/10.1145/1518701.1518895.

[184] Jian Tang et al. "Line: Large-scale information network embedding". In: *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2015, pp. 1067–1077.

[185] *The State of Data Science and Machine Learning*. 2017. URL: https://www.kaggle.com/surveys/2017.

[186] *Towards ML Engineering: A Brief History Of TensorFlow Extended (TFX)*. accessed 2020-09.

[187] TransmogriFAI. *TransmogrifAI*. Website. 2020. URL: https://github.com/salesforce/TransmogrifAI.

[188] Michelle Vaccaro and Jim Waldo. "The Effects of Mixing Machine Learning and Human Judgment". In: *Commun. ACM* 62.11 (Oct. 2019), pp. 104–110. ISSN: 0001-0782. DOI: 10.1145/3359338. URL: https://doi.org/10.1145/3359338.

[189] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

[190] Jan N Van Rijn and Frank Hutter. "Hyperparameter Importance Across Datasets". In: (2018), p. 10. DOI: 10.1145/3219819.3220058. arXiv: 1710.04725v2. URL: https://doi.org/10.1145/3219819.3220058.

[191] Joaquin Vanschoren. *Meta-Learning: A Survey*. Tech. rep. 2018. arXiv: 1810.03548v1.

[192] Joaquin Vanschoren et al. "OpenML: Networked Science in Machine Learning". In: *SIGKDD Explorations* 15.2 (2013), pp. 49–60. DOI: 10.1145/2641190.2641198. URL: http://doi.acm.org/10.1145/2641190.2641198.

[193] Manasi Vartak and Samuel Madden. "MODELDB: Opportunities and Challenges in Managing Machine Learning Models". In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 16–25. URL: http://sites.computer.org/debull/A18dec/p16.pdf.

[194] Manasi Vartak et al. "MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis". In: *Proceedings of the 2018 ACM International Conference on Management of Data*. 2018.

[195] Manasi Vartak et al. "Supporting fast iteration in model building". In: *NIPS Workshop LearningSys*. 2015.

[196] Daisy Zhe Wang et al. "Hybrid in-database inference for declarative information extraction". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 517–528.

[197] Dakuo Wang et al. "Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI". In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (Nov. 2019). DOI: 10.1145/3359313. URL: https://doi.org/10.1145/3359313.

[198] Qianwen Wang et al. "ATMSeer: Increasing Transparency and Controllability in Automated Machine Learning". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, pp. 1–12. ISBN: 9781450359702. DOI: 10.1145/3290605.3300911. URL: https://doi.org/10.1145/3290605.3300911.

[199] Daniel Karl I. Weidele et al. *AutoAIViz: Opening the Blackbox of Automated Artificial Intelligence with Conditional Parallel Coordinates*. 2019. arXiv: 1912.06723 [cs.LG].

[200] Markus Weimer, Tyson Condie, Raghu Ramakrishnan, et al. "Machine learning in ScalOps, a higher order cloud computing language". In: *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*. Vol. 9. 2011, pp. 389–396.

[201] *What are Azure Machine Learning pipelines?* accessed 2020-11.

[202] Wikipedia contributors. *One-hot — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-September-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=One-hot&oldid=975049657.

[203] Kanit Wongsuphasawat et al. "Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 1–12. DOI: 10.1109/TVCG.2017.2744878.

[204] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), p. 19.

[205] Brett Wujek, Patrick Hall, and Funda Güneș. *Best Practices for Machine Learning Applications*. Tech. rep. 2016, pp. 1–23.

[206] Doris Xin et al. "Accelerating Human-in-the-loop Machine Learning: Challenges and Opportunities (Vision Paper)". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. DEEM'18. ACM, 2018.

[207] Doris Xin et al. "Helix: Accelerating Human-in-the-loop Machine Learning (Demo Paper)". In: *Proceedings of the VLDB Endowment* (2018).

[208] Doris Xin et al. "Helix: Holistic Optimization for Accelerating Iterative Machine Learning". In: *Technical Report http://data-people.cs.illinois.edu/helix-tr.pdf* (2018).

[209] Doris Xin et al. "Helix: holistic optimization for accelerating iterative machine learning". In: *Proceedings of the VLDB Endowment* 12.4 (2018), pp. 446–460.

[210] Doris Xin et al. "How Developers Iterate on Machine Learning Workflows–A Survey of the Applied Machine Learning Literature". In: *KDD IDEA Workshop* (2018).

[211] Doris Xin et al. "Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference*. ACM, 2021. DOI: 10.1145/3448016.3457566.

[212] Doris Xin et al. "Whither AutoML? Understanding the Role of Automation inMachine Learning Workflows". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2021. DOI: 10.1145/3411764.3445306.

[213] Qian Yang et al. "Grounding Interactive Machine Learning Tool Design in How Non-Experts Actually Build Models". In: *Proceedings of the 2018 Designing Interactive Systems Conference*. DIS '18. Hong Kong, China: Association for Computing Machinery, 2018, pp. 573–584. ISBN: 9781450351980. DOI: 10.1145/3196709.3196729. URL: https://doi.org/10.1145/3196709.3196729.

[214] Mihalis Yannakakis. "On a class of totally unimodular matrices". In: *Mathematics of Operations Research* 10.2 (1985), pp. 280–304.

[215] Jia Yu and Rajkumar Buyya. "A taxonomy of workflow management systems for grid computing". In: *Journal of Grid Computing* 3.3-4 (2005), pp. 171–200.

[216] Yuan Yu et al. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In: *OSDI*. Vol. 8. 2008, pp. 1–14.

[217] Matei Zaharia. *Introducing MLflow: an Open Source Machine Learning Platform*. https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html. Accessed October 8, 2018.

[218] Matei Zaharia et al. "Accelerating the Machine Learning Lifecycle with MLflow." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 39–45.

[219] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012.

[220] Amy X Zhang, Michael Muller, and Dakuo Wang. "How do data science workers collaborate? roles, workflows, and tools". In: *Proceedings of the ACM on Human-Computer Interaction* 4.CSCW1 (2020), pp. 1–23.

[221] Ce Zhang. "DeepDive: A data management system for automatic knowledge base construction". PhD thesis. Citeseer, 2015.

[222] Ce Zhang, Arun Kumar, and Christopher Ré. "Materialization optimizations for feature selection workloads". In: *ACM Transactions on Database Systems (TODS)* 41.1 (2016), p. 2.

[223] Martin Zinkevich. *Rules of Machine Learning: Best Practices for ML Engineering*. 2017.

[224] Martin Zinkevich et al. "Parallelized stochastic gradient descent". In: *Advances in neural information processing systems*. 2010, pp. 2595–2603.