Mining Software Artifacts for use in Automated Machine Learning

by

José Pablo Cambronero Sánchez

B.A., University of Pennsylvania (2011)
M.S., New York University (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by .....
Martin C. Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Mining Software Artifacts for use in Automated Machine Learning

by

José Pablo Cambronero Sánchez

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Successfully implementing classical supervised machine learning pipelines requires that users have software engineering, machine learning, and domain experience. Machine learning libraries have helped along the first two dimensions by providing modular implementations of popular algorithms. However, implementing a pipeline remains an iterative, tedious, and data-dependent task as users have to experiment with different pipeline designs. To make the pipeline development process accessible to non-experts and more efficient for experts, automated techniques can be used to efficiently search for high performing pipelines with little user intervention. The collection of techniques and systems that automate this task are commonly termed automated machine learning (AutoML).

Inspired by the success of software mining in areas such as code search, program synthesis, and program repair, we investigate the hypothesis that information mined from software artifacts can be used to build, improve interactions with, and address missing use cases of AutoML. In particular, I will present three systems – AL, AMS, and Janus – that make use of software artifacts. AL mines dynamic execution traces from a collection of programs that implement machine learning pipelines and uses these mined traces to learn to produce new pipelines. AMS mines documentation and program examples to automatically generate a search space for an AutoML tool by starting from a user-chosen set of API components. And Janus mines pipeline transformations from a collection of machine learning pipelines, which can be used to improve an input pipeline while producing a nearby variant. Jointly, these systems and their experimental results show that mining software artifacts can simplify AutoML systems, make their customization easier, and apply them to novel use cases.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Completing my PhD has been one of the most challenging and rewarding things I have done to date. It would have been impossible to successfully complete it without the contributions and support of many, both at MIT and elsewhere. I would like to take a moment to highlight some of these wonderful individuals.

I owe my advisor, Martin Rinard, a huge debt of gratitude. During my time at MIT he provided me with guidance on research direction, professional and life advice, as well as support for the times when I was struggling with my own research motivation. He challenged, and in doing so strengthened, many of the initial research ideas I had. I cannot imagine having had a better guide for this journey.

I also would like to thank the other members of my thesis committee: Armando Solar-Lezama and Saman Amarasinghe. Both Armando and Saman raised interesting questions that will continue to drive this research direction. I also want to thank them, because during my time at MIT, they were always kind and inquisitive about my (and other students') research.

The PAC group, with weekly lunches and casual conversations, provided not only excellent colleagues but also kind individuals. Special thanks go to both current and former members: Sara Achour, Jiasi Shen, Shivam Handa, Yichen Yang, Kai Jia, Charles Jin, Austin Gadient, Malavika Samak, Thurston Dang, Nikos Vasilakis, Deokhwan Kim, Fan Long, Phillip Stanley-Marbell, Jeff Perkins, Mary McDavitt, and Stelios Sidiroglou-Douskos.

I would like to thank two former PAC members in particular. Phillip Stanley-Marbell provided me with a lot of support and encouragement during my first year of graduate school. His advice and encouragement turned an otherwise intimidating first year into a wonderful dive into research in areas I had never explored. Similarly, I am extremely thankful to Jürgen Cito. Jürgen was one of the first people to provide words of encouragement when I was frustrated with my research direction, and he highlighted many of the positives (which after working on a topic for a while can become hard to see). Jürgen not only contributed encouragement: he was also a direct contributor to two of the systems presented in this thesis (AMS and Janus).

I would like to thank Raul Castro Fernandez. Raul, like Jürgen, was one of the earliest supporters of this research direction. He has a unique ability for providing feedback in the most encouraging way I have ever encountered. His enthusiasm for ideas is absolutely contagious and I am lucky to have had the opportunity of collaborating with him.

I want to thank many of the friends I made during my time at MIT. Finishing my PhD would have been unlikely had I not had the company and support of these friends. Thank you: Jack Feser, Will Stephenson, Sam DeLaughter, Maz Abulnaga, Eric Atkinson, Ben Sherman, Jess Ray, Tarfah Alrashed, Joana M.F. da Trindade, Caris Moses, Darsh Shah, and Micah Smith. I owe a special thanks to Micah, who in addition to being a good friend is also a phenomenal researcher and throughout my PhD has provided priceless feedback, as well as made key contributions to Janus (one of the systems presented in this thesis). I also owe Darsh for his support of my research ideas and for his (voluntary!) help with a dry-run of my defense.

I would also like to thank some individuals that played key roles before my time at

MIT. My first job out of undergrad turned out to be a pivotal moment in my life. Up to that point, I thought my path would continue to be in economics/finance. Through a turn of luck, Vishy Tirupattur and Jim Egan hired me into a research group. I then sat next to Mike Jansen, during a summer, who introduced me to Jeff Borror. All four individuals encouraged my interest in programming, which quickly devolved into a near-obsession with computer science. Vishy and Jim were incredibly tolerant of my random attempts to automate tasks. Jeff gave me priceless advice, encouraged me to pursue graduate studies, and has since become a dear friend.

I would like to thank Eva Rose and Kris Rose. Eva and Kris exposed me to my first compilers course and not only shared their own excitement about the field, they also actively encouraged me to pursue my own research. Kris took time to discuss different grad schools and directions with me, and provided invaluable feedback. He will be dearly missed.

I also would like to thank Dennis Shasha, who supervised my research while at NYU. Dennis gave me the freedom to explore and grow as a researcher working in a new field.

I have had the opportunity to collaborate with great people on a healthcare project as well. I would like to thank Limor Appelbaum for her advice and collaboration. I also owe a thanks to the undergraduate and MEng students who I have worked with: Jennifer McCleary, Alex Berg, Lori Zhang, and Thomas Xiong.

I would have never gotten a chance to attend MIT, nevermind finish my PhD, had it not been for my family. My mother, Carmen Sánchez, has provided a shining example for the kind of person I would like to be: kind, disciplined, and excited to learn new things. My father, Efraín Cambronero, has also been a strong supporter of education and encouraged us to pursue our fields of interest. My brothers, David and Andrés, have both been role models of mine. David has motivated me with his passion in pursuing his own interests as an art historian. Andrés has long been an example of a thoughtful and disciplined statistician.

I want to provide a special thanks to the most important person in my life: my wife, Rebecca Earnest. Rebecca has shared with me the most meaningful events of my life: university, grad school, marriage. It would have been impossible to go through this journey without her support. Rebecca provided words of encouragement and has been my biggest cheerleader. I now have very big shoes to fill by doing the same for her on her own PhD journey.

I also need to thank our cat, Don Vincenzo. It's hard to feel stressed when you pet him, and as such, he's likely extended my life through his sheer existence.

I owe a special thanks to all the data scientists, software developers, and researchers who have made their own systems and data publicly available. Particularly, I want to thank Kaggle and its users, the scikit-learn community and developers, the Autosklearn developers, and the TPOT developers. Their contributions have been invaluable for my thesis research.

I would like to dedicate this thesis to the memory of my grandparents: Amado Sánchez and Flor de Maria Arredondo. My grandparents have been the single largest influence on my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Context

Building and maintaining supervised machine learning systems is increasingly important in fields as varied as medicine [93], geology [98], and finance [35]. Successfully building such systems requires a combination of domain, machine learning, and software engineering experience. To address the latter two dimensions, popular machine learning libraries such as Python's scikit-learn [89] provide modular implementations of popular algorithms encapsulated in re-usable and configurable classes or functions. The user's task of building a supervised learning pipeline then is formulated as choosing, composing, and configuring library components. These modular components simplify the process of building a pipeline, however, the search space of possible pipelines is still too large to exhaustively enumerate. Furthermore, the optimal pipeline design will vary based on the underlying task and dataset. In practice, this means that developing machine learning pipelines tends to be an interactive, difficult, and tedious process, where developers design, implement, and empirically validate the performance of alternative pipelines. Furthermore, the extent to which developers explore design choices, ranging from the choice of components to the fine-tuning of hyperparameters, varies based on their task, machine learning experience, and timeline constraints.

To make the process of pipeline implementation accessible to non-experts and more efficient for experts, automated procedures can be employed to automatically explore the design space of pipelines. These search techniques are broadly termed *automated machine*

learning (AutoML). The general approach for these systems is to iteratively generate candidate pipelines (or choose from a portfolio of pre-defined pipelines), empirically estimate performance using some form of cross-validation (the results of which may be used to propose new candidates), and return one or more high performing pipelines to the user, subject to a computation time budget.

Past work in AutoML has explored hyperparameter optimization, as well as transformation- and model-selection. Hyperopt [13] uses structured search spaces to model pipeline choices and a sequential-sampling approach based on a generative model. TPOT [82] employs genetic programming to automatically produce tree-structured classification and regression pipelines composed of scikit-learn [89] operators. Autosklearn [41] uses sequential model-based algorithm configuration (SMAC) [57] to generate scikit-learn pipelines and their hyperparameter settings. ReinBo [109] uses reinforcement learning to generate pipeline candidates and Bayesian optimization to tune their hyperparameters, propagating performance results back to the reinforcement learning system. MLBazaar [107] builds up an AutoML system through a library of composable operators with a clean and unified interface. However, none of these approaches exploit information found in *existing* software artifacts such as code examples or documentation.

"Software artifacts" encompass a broad set of outputs or by-products of the process of software engineering. Such artifacts include source code repositories, documentation, concrete program executions, online question answering forums, requirements documents, and developer issues, among others. In practice, software artifacts provide structured, semi-structured, and unstructured data that reflect collective knowledge about different software tasks. The increased availability of such artifacts, facilitated in large part through large-scale online repositories like GitHub [45], has driven an interest in systematically extracting re-usable information from them, which can be used to address related development tasks. Software engineering and programming languages researchers have successfully mined existing software artifacts to tackle varied problems, including searching for code using natural language [18], synthesizing automated program repair templates from developer changes [74], formalizing usage specifications from documentation [127], synthesizing programs by combining low-level statements and high-level idioms [106], synthesizing

programs from trace-based demonstrations [125], and mapping functionality between two different APIs [87]. These techniques share common characteristics such as replacing system-designer heuristics or user effort with mined/learned information.

1.1.1 Thesis Hypothesis

It is with this context in mind, that I present the following hypothesis for investigation. **Information mined from software artifacts can be used to build, improve interactions with, and address missing use cases of automated machine learning systems.** To investigate this hypothesis, I developed three systems – AL, AMS, and Janus – each of which mines information from software artifacts and uses it to tackle different challenges in automated machine learning. AL was designed to investigate the possibility of learning component selection and composition from example programs. AMS investigates the extent to which we can automatically generate an AutoML search space from a user-chosen set of API components. Janus investigates the hypothesis that we can mine pipeline transformations from a collection of machine learning pipelines and use these to improve the performance of existing pipelines by transforming them to *nearby* variants.

1.2 AL: Mining program executions

Most AutoML systems construct candidate pipelines automatically by composing components – modular implementations of popular algorithms – available in dedicated machine learning libraries, such as Python's scikit-learn [89]. The choice and order of components, along with their hyperparameter configurations, is then explored by the system's underlying search technique.

System Design We developed AL to investigate the hypothesis that AutoML systems can implement component selection and composition by learning from a corpus of example programs that implement their own machine learning pipelines. In particular, AL exploits information available at runtime about the choice of components, the order they are composed in, and the characteristics of the data (at the time of component application). To

collect this information, we instrument a collection of approximately 500 Python programs that use scikit-learn or XGBoost, both popular Python machine learning libraries. We record calls to these two target libraries, along with meta-features that summarize the input data to that call. We also record data dependencies in order to extract the order of component composition.

AL uses this dynamic trace information to construct two models for pipeline completion, conditioned on a partial pipeline and the state of the input dataset. To capture the state of the input dataset, AL relies on a set of 26 meta-features that capture key characteristics such as the fraction of missing values, the frequency of column types, the dimensions of the input table, among others. Given a partial pipeline, AL predicts the next component to be added into the sequential pipeline as a function of the prior k components and the latest state of the dataset. This incremental completion is then used in a beam search to incrementally build up candidate pipelines, all of which use the default hyperparameters defined in the underlying library.

Experimental Results We compared AL to two existing AutoML tools – *autosklearn* [41] and *TPOT* [82] – on a total of 31 datasets collected from the original *autosklearn* paper, the original *TPOT* paper, the data science website Kaggle [46], and the multi-target regression benchmarking suite Mulan [113].

Our experimental results show that AL can produce pipelines with comparable predictive performance to *autosklearn* or *TPOT* for 17 of the 21 datasets from their respective papers. For most of these datasets, AL can generate these pipelines on average within 5 minutes, compared to a search budget of one or two hours for each of the competing systems. For the 10 datasets collected from Kaggle and Mulan, we show that AL is the only system of the three to generate pipelines without any additional user intervention. In particular, both *autosklearn* and *TPOT* are missing key preprocessing components from their search space. Examples of these preprocessing steps include transforming string columns into a numeric encoding. Not applying such transformations results in runtime errors for the Kaggle datasets. For the Mulan datasets, both *autosklearn* and *TPOT* perform additional input validation that raises an exception for multi-target outputs, despite the fact that there are components in their under-

lying search spaces that can handle multi-target regression. In contrast, AL's search space contains components (observed in example programs) that support multi-target outputs and does not perform any additional input validation, and so can successfully generate pipelines.

We also compared the predictive performance of pipelines generated by AL to those generated by two ablated system versions. *Sklearn-XGBoost-Random* randomly samples components from the entirety of the target libraries scikit-learn and XGBoost, while *AL-Random* only considers components that AL encounters in example programs but chooses and composes them uniformly at random and prunes pipelines uniformly at random during search. Our experimental results show that for 19 of 21 datasets the average F1 score for the top 10 pipelines generated by *AL-Random* is higher than for those pipelines generated by *Sklearn-XGBoost-Random*. Furthermore, we find that for 19 of 21 datasets the full AL system improves over the average pipeline scores produced by *AL-Random*.

We submitted predictions from AL-generated pipelines for three different open Kaggle competitions. At the time of submission, the predictions submitted out-performed 29%, 51.6% and 91.5% of users' submissions.

We also present detailed system design experiments. In particular, we evaluate the overhead imposed on the original programs by AL's instrumentation during the offline pipeline collection phase. We also evaluate whether searching for pipelines exclusively as a function of prior components compared to AL's extended model with data meta-features impacts search time. Finally, we present results on the distribution of components in AL's generated pipelines.

1.3 AMS: Mining documentation and code examples

Automatically generating machine learning pipelines frees users from the tedious (and difficult) task of manually designing and evaluating pipeline candidates. However, popular techniques typically assume that the user provides no input to the search beyond providing the initial input dataset. In effect, the search is solely focused on maximizing the performance criteria selected, subject to a time constraint. In an effort to provide users with more control, interfaces increasingly support user-defined search spaces in the form of a set

of eligible components, and hyperparameter search spaces for each [82]. However, to use a custom search space the user must fully specify the set of components and hyperparameter search spaces manually. Defining this search space manually negates a core benefit of AutoML: the user need not be a machine learning expert.

System Design We developed a system, named AMS, to investigate the hypothesis that such AutoML search spaces can be automatically constructed based on a small amount of user input. Furthermore, we investigate the extent to which these search spaces, when paired with different pipeline sampling algorithms, produce high performing pipelines, while generating candidate pipelines that reflect the influence of the user's initial set of API components.

AMS starts the search space generation process by taking a set of user-chosen API components as a seed. We term this set the search space's *weak specification*. It is weak in the sense that it is incomplete along several key dimensions: it is potentially missing other relevant API components, does not specify what hyperparameters to tune, and does not specify what values hyperparameters can take on. To address these shortcomings, AMS mines information from existing software artifacts.

First, AMS mines the API's documentation to identify components that are functionally related – based on the information retrieval metric BM25 [97] – to those that the user specified and can be added to the current set of components as alternatives. Next, AMS mines a corpus of programs that use our target scikit-learn library to identify components that tend to co-occur, and thus may complement each other. Complementary components can potentially improve the performance of a pipeline when used jointly. For example, different normalization components may improve the performance of a downstream classifier. To determine whether components truly tend to co-occur, in contrast to co-occurring due to random chance, AMS makes use of a probabilistic metric: normalized pointwise mutual information (NPMI) [15]. AMS uses the same program corpus to identify hyperparameters that are commonly set by users, along with common values. To identify hyperparameters and their values, AMS makes the assumption that the values they take on are independent and discrete and collects the set of values observed in API constructor calls in the program corpus.

The extended set of components, along with their hyperparameter search spaces, can

then be paired with an existing search procedure to sample candidate pipelines. In particular, the search procedure can choose components from the extended set produced by AMS and can tune the subset of hyperparameters identified by AMS by choosing from the set of values AMS defines. Different sampling procedures may choose to compose components, and their configurations, differently. In our implementation of AMS, we provide two different search procedures. We implemented a custom random search procedure that samples components (and their hyperparameter configurations) independently and composes them sequentially up to a length bound. We can also sample pipelines using a genetic programming-based search by feeding the search space generated directly to an existing genetic-programming-based AutoML tool: TPOT.

Experimental Results We implemented AMS to target the popular Python machine learning library scikit-learn. We compared the pipelines produced by searching in AMS-generated search spaces to three alternative baselines that would allow a user to restrict the search space (manually) and incorporate varying degrees of sophistication. First, we consider a baseline that directly applies (in a pre-defined order) the weak specification API components, with default hyperparameters, as a pipeline. Second, we consider a baseline that applies a search procedure over the weak specification API components, with default hyperparameters, reflecting different subset choices and composition orders. Finally, we consider a baseline that defines an expert-based hyperparameter search space for each component in the weak specification and then applies a search procedure over this extended search space definition.

We considered a total of 15 weak specifications, generated by combining popular pre-processors and classifiers. We generated a search space from each weak specification using AMS, and then compared the performance of a search procedure sampling pipelines from that space compared to the baselines outlined previously. We carried out these experiments on 9 different datasets from the TPOT paper. To compare pipelines produced across competing approaches, we used the concept of a *win*. A pipeline wins when it obtains the highest score on a specification/dataset combination, and satisfies a minimum predictive score difference to rule out comparable performance.

We find that AMS-generated search spaces produce more wins than our baselines when using two different search procedures. When using genetic programming as a search procedure, the search spaces produced by AMS result in 38 wins compared to 12 under the weak specification extended with an expert-defined hyperparameter space. When using random search as a search procedure, the search spaces produced by AMS result in 41 wins compared to 14 wins under the weak specification extended with an expert-defined hyperparameter space.

Qualitatively, we find that the distributions of top 10 components in the candidate pipelines generated by sampling from AMS-generated search spaces reflect the influence of the initial weak specifications. For example, component distributions appropriately include functionally-related and complementary API components.

We also include extensive system design experiments. In particular, we evaluate the precision of our NPMI-based complementary component mining approach and the precision of our BM25-based functionally related component retrieval. We also characterize the diversity of hyperparameters tuned and their values in the programs in our example corpus. Finally, we also evaluate the extent to which varying the program corpus size affects the sets of complementary components mined and sets of hyperparameters and values observed.

1.4 Janus: Mining pipeline transformations

When using a traditional AutoML system, a user provides their input dataset and a performance metric to optimize. The AutoML system then executes a search and returns one or more pipelines to the user. However, this setting does not account for users who already have a pipeline and want to improve their *existing* implementation. While such a user could run an AutoML search, there is no guarantee that the resulting pipeline bears any resemblance to their current pipeline, which may reflect properties such as domain knowledge or resource constraints. In such a setting, we would like to identify alternative pipelines with higher performance but that are similar to the user's original pipeline.

System Design We developed Janus to investigate the hypothesis that we can mine nearby transformations from a large collection of machine learning pipelines to identify changes that are associated with higher performance, and which can be applied to an input pipeline to produce higher performing variants that resemble the original pipeline.

The prospect of mining transformations from a collection of pipelines is inspired by related work in automated program repair, where productive patches can be extracted from paired program versions [74, 99, 100]. In Janus, we collect pairs of pipelines and identify changes associated with pairs where there is a performance differential.

Janus starts by collecting a large number of machine learning pipelines. To collect these pipelines efficiently, Janus capitalizes on the insight that existing AutoML tools are already effective at exploring different parts of the pipeline design space, producing both good *and* bad pipelines. Janus relies on this performance differential between similar pipelines to identify productive transformations. From this collection of pipelines, Janus identifies pairs of pipelines that are nearby in tree-edit distance and for which there is a performance differential on the same dataset. We term these pairs of trees d-repairs, as the higher performing pipeline in the pair must be within d tree-edit operations of the lower performing pipeline. To efficiently identify such pairs without computing the exact tree-edit distance for all candidate pairs, Janus employs a linear-time approximate distance based on a vector representation of the pipelines, and only computes the exact tree-edit distance on a subset of pipelines that rank closely based on their approximate distance.

From a collection of d-repairs, Janus extracts the sequence of tree-edit operations required to transform the lower performing pipeline into its higher performing counterpart. While a sequence of edit operations provides the steps necessary to transform the *specific* lower performing pipeline into the *specific* higher performing pipeline, they need to be adapted to apply to a new input pipeline. To adapt them, Janus first converts the generic edit operations into a typed-abstraction termed local structural rules (LSR), which 1) distinguish between component and hyperparameter nodes in the pipeline tree, and 2) incorporate checks on neighboring nodes to determine whether a rule is applicable (along with the expected change in the output pipeline structure when the rule is applied). Janus then summarizes the collection of LSRs extracted by heuristically picking a concrete LSR

for each group of rules that share a common key, defined as the type of edit operation, the label of the original target node, a subset of neighbor labels, and the label of the resulting node after application.

Given that multiple transformations may apply to a new input pipeline, Janus must decide how to prioritize these rules. To rank rules Janus computes the joint probability of a given transformation's key and the node where it was applied in the corpus of pairs collected. Given a new input pipeline, Janus greedily applies transformations, ranked by their joint probability, and returns the first edit that produces a tree that is 1) within the tree-edit distance bound specified by a user, 2) can be compiled to a valid Python object, and 3) can be executed on a small sample of the input dataset without raising any runtime exceptions.

Experimental Results We implemented Janus for scikit-learn classification pipelines and compare its performance to three different baseline approaches. We considered a baseline that randomly mutates an input pipeline, a baseline that randomly samples a transformation mined by Janus, and a meta-learning inspired baseline that produces up five randomly mutated variants and then returns the pipeline with the highest expected score. To predict a pipelines' expected score the last baseline uses a model that regresses a pipeline's F1 score as a function of a pipeline's vector representation.[1]

To produce a corpus of pipelines, we ran TPOT on 9 different datasets. For each dataset, we randomly sample 100 different pipelines and use these as our evaluation's input pipelines. We then evaluate each system's ability to produce d-repairs for these 900 input pipelines, To compare outcomes, we say a pipeline is improved if its modification results in an F1 score increase of at least 0.01. We say a pipeline was hurt if its modification results in an F1 score decrease of at least 0.01. When carrying out experiments, we evaluate a transformed pipeline's performance using 5-fold cross-validation. Additionally, we take a leave-a-dataset-out approach, where we blind all approaches to the pipelines associated with the same dataset for which we are evaluating transformations.

Our experimental results show that Janus can improve $16\% - 42\%$ of the pipelines across our test datasets, more than baseline approaches in 7 of our 9 datasets. Our experimental

[1] the same representation that Janus uses to compute approximate distance

results also show that Janus hurts fewer pipelines than the baselines in 6 of our 9 datasets. The distribution of pipeline score increases for pipeline improvements shows that Janus results in score improvements comparable to the random mutation and random Janus baselines but lower than those induced by the meta-learning inspired baseline. However, when pipelines are hurt, Janus reduces pipeline performance by substantially less than the alternative approaches. When both Janus and the meta-learning baseline improve *the same* pipeline, we find that the amount by which the F1 score increases is comparable (with no statistically significant difference). Finally, we find that for pipelines improved, the modified pipelines produced by Janus are closer to the original input pipelines than those produced by the baseline approaches.

We also carry out additional system design experiments. In particular, we empirically validate that the distance approximation used by Janus when collecting d-repairs dominates a random sampling approach and closely tracks the performance of an exact tree-edit distance approach. We also compute the rule corpus reduction resulting from the heuristic summary of LSRs based on a key. We also show that the underlying pipeline corpus plays a significant role in the effectiveness of the transformations mined. When we replace the corpus with one generated by a search procedure that constructs randomly sampled sequential pipelines, Janus no longer outperforms the meta-learning baseline on our datasets.

1.5 Contributions

The contributions of this thesis are centered on the design, implementation, and evaluation of the three systems used to investigate the thesis hypothesis presented in Section 1.1.1.

- **AL.** We present a new automated machine learning system, AL, that is the first to extract supervised machine learning pipelines from program traces and use these to learn how to generate pipelines for new datasets. We present the pipeline extraction and learning procedures used by AL. We present an evaluation that shows that the simple search procedure implemented by AL produces pipelines with comparable performance to two existing AutoML systems and generalizes to a collection of datasets where the other systems fail to run without modifications. Fundamentally,

AL demonstrates that learning component selection and composition from programs traces can be done successfully and presents a more flexible (and general) approach to designing the component search space for AutoML systems.

- **AMS.** We introduce a novel approach for automatically generating a pipeline search space for an AutoML tool by augmenting a set of API components identified by a user. We introduce the functionally-related component retrieval and complementary component mining algorithms that allow us to extend the set of API components, where the former leverages the API documentation and the latter a corpus of existing programs that exercise the target API. We also show how this corpus of programs can be mined to identify popular hyperparameters and their values, forming the basis for a simple, counting-based approach to constructing hyperparameter search spaces. Our evaluation shows that AMS can produce search spaces with high performing pipelines and that candidates generated from these spaces qualitatively reflect the original input set of components. Fundamentally, AMS shows that partial user input can be augmented with information from documentation and example programs to automatically generate search space definitions.

- **Janus.** We develop procedures for identifying pipeline pairs for rule mining using an efficient approximate tree-edit distance, extracting and summarizing transformations in the form of rules, and generating transformed pipelines. We introduce an abstraction, local structural rules, to represent typed edit operations with machine learning pipeline specific semantics. We develop a rule prioritization algorithm based on the joint probability of a rule and the target node where it is to be applied. We implemented and evaluated a version of Janus that transforms pipelines implemented using scikit-learn [89], a popular Python machine learning library. Our experimental results show that a higher fraction of transformed pipelines generated by Janus improve predictive performance, and a lower fraction hurt predictive performance, compared to our baselines. Fundamentally, Janus shows that we can mine transformations from pairs of similar machine learning pipelines and that these transformations can be systematically used to produce nearby higher performing variants of input pipelines.

Jointly, these systems and their experimental results provide support to the hypothesis that we can mine information from software artifacts and use it effectively in the design of and interaction with automated machine learning systems. Particularly, we can replace system designer choices, such as the set of pre-defined components in a search space, with learned alternatives that enable more flexibility. We can bridge the gap between non-expert users and systems that require expertise for customization by automatically augmenting user information. And we can leverage prior pipeline designs and outcomes to identify productive changes for software evolution. At a high level, software artifacts reflect diverse user development experiences, ranging from experts to beginners, and incorporating information mined from such artifacts can transfer some of that diversity and flexibility to automated procedures such as AutoML systems.

1.6 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 provides key background content for understanding the core of the thesis. This background focuses on supervised machine learning, machine learning pipelines (as a composition of API components), and automated machine learning. Chapter 3 presents the first contribution of this thesis: the approach and evaluation underlying the AL system. Chapter 4 discusses AMS, our system for automatically generating AutoML search spaces. Chapter 5 presents the final contribution of this thesis: Janus, a system that automatically mines and applies machine learning pipeline transformations to produce higher performing variants. Chapter 6 provides a survey of related work in AutoML and mining of software artifacts. Finally, Chapter 7 concludes with high-level remarks and takeaways.

Chapter 2

Background

In this chapter, I will introduce background concepts that will be used throughout my thesis.

2.1 Supervised Machine Learning

We focus our discussion of supervised machine learning on regression and classification tasks. Many practical tasks can be framed as regression or classification, such as predicting stock prices [35], early pancreatic cancer diagnoses [5], and identifying fraudulent credit card transactions [8].

2.1.1 Tabular and Vector Inputs

We assume a setting where the user provides an input table, X, consisting of n rows, each of which is an m-tuple in $(D_1 \times \cdots \times D_m)$, where D_i is the type of the ith column and is typically integer, string, float, or boolean. We let the type of X be denoted as $\textsc{Table}[D_1 \times \cdots \times D_m]$. The column values in X are commonly termed covariates, attributes, or features. The user also provides an accompanying (column) vector of target values $y \in D_y^{n \times 1}$, one for each row in X, where the data type D_y is typically string or integer (for classification) or float (for regression). We let the type of y also be denoted as $\textsc{Vector}[D_y]$. For convenience we subscript variables to indicate the value at the corresponding index. For example, x_i denotes the ith row in X and y_i denote the ith entry in y. The dataset

$(X,\ y)$ can be further split into training, $(X_{\text{train}},\ y_{\text{train}})$, and test, $(X_{\text{test}},\ y_{\text{test}})$. Such splitting can be further generalized to k-fold cross validation, where the dataset is split into k (approximately equally sized) portions, each of which is used as a test set once (and remaining portions of the dataset are used as the training set) [42].

2.1.2 Classification and Regression

The goal of supervised machine learning is to learn a function $f : (D_1 \times \cdots \times D_m) \to D_y$ that can *predict* the target values of a set of observations such that it maximizes predictive performance as measured by an evaluation function $e : D_y \times D_y \to \mathbb{R}$ that computes the quality of the predictions. Often e is framed as a loss function, for example the squared error loss function $(y_i - f(x_i))^2$ [42], such that a better f will produce a lower error.

Typically f has parameters Θ that can be learned from the dataset, and our goal is to choose values to minimize the errors produced. The process of learning these parameters from the training split of the data is commonly referred to as *fitting* f to the training data. Let FIT have the type signature

$$\text{TABLE}[D_1 \times \cdots \times D_m] \times \text{VECTOR}[D_y] \to (D_1 \times \cdots \times D_m \to D_y)$$

where Θ values are obtained by solving the optimization problem below

$$\underset{\Theta}{\operatorname{argmin}} \frac{1}{|X_{\text{train}}|} \sum_{x_i \in X_{\text{train}},\ y_i \in y_{\text{train}}} e(f_\Theta(x_i),\ y_i)$$

2.1.3 Data Transformations

It is often the case that datasets need to be transformed to make them compatible with different algorithms. For example, a developer interested in using a support vector machine to predict target values when there are missing values in X will need to either remove the covariates with missing values or predict a replacement value (a task known as imputation [114]). Similarly, if the dataset has string covariates, but the user wishes to employ a learning algorithm designed exclusively for numeric data, they will have to either remove

these covariates or re-encode them to a numeric representation.[1]

In general, we refer to such transformations as *data transformations*. We focus on data transformations that depend on a single input table, but in a more general setting data transformations can be a function of more than one table (e.g. join transformations). However, such transformations traditionally occur in the *data wrangling* portion of the analysts workflow, which we do not focus on.

Formally, let data transformations be functions of the form

$$D_1 \times \cdots \times D_m \to D'_1 \times \cdots \times D'_j$$

where datatype D_i may be different from datatype D'_i and m may be different from j. In other words, a data transformation may change the number and types of columns in the input table.

Note that a data transformation, similarly to regressors or classifiers, may have parameters that can be computed from the dataset. For example, an imputation transformation may compute a mean from the non-missing values available in X_{train} and use that value to fill in missing entries in X_{test}. As such, we define the function FIT for data transformations to have the signature below:

$$\text{TABLE}[D_1 \times \cdots \times D_m] \to (D_1 \times \cdots \times D_m \to D'_1 \times \cdots \times D'_j)$$

2.1.4 Supervised Learning as Pipelines

We define a supervised learning pipeline to be the sequence of zero or more data transformations, followed by a regressor or classifier that jointly implement the prediction function f. We refer to the process of fitting the data transformations and the regressor or classifier on the training data as *fitting* the pipeline.

[1]Third-party libraries may perform data transformations implicitly for users' convenience.

2.2 Machine Learning Libraries

While software developers may find it useful to re-implement popular data transformations or regression/classification algorithms, in practice developers make use of popular machine learning libraries to implement their pipelines. Machine learning libraries provide stand-alone algorithms, or their key components, as classes or functions[2] that developers can use to implement pipelines for their tasks. For example, users targeting time series may use statsmodels [104] time series analysis module, while those interested in anomaly (i.e. outlier) detection may use PyOD [126]. Meanwhile, users who are interested in using neural networks for their task can use PyTorch [88], JAX [16], and Keras [25], among others.

2.2.1 scikit-learn

While the ideas presented in this thesis are applicable to different machine learning libraries, my implementations and evaluation are focused on systems that make use of scikit-learn as their machine learning library. Scikit-learn [89] is a popular machine learning library for the Python language, focused on classical machine learning algorithms.[3] As of April 2021, scikit-learn has been downloaded 178,625,155 times from PyPi (a major Python repository) in the last 365 days, has had 115 releases[4], has 1,946 GitHub contributors, is a dependency for 200,130 other GitHub repositories, and has 45,100 GitHub stars.

Scikit-learn implements popular algorithms, such as decision trees and random forests, and wraps efficient learning tools such as LIBSVM [22] and LIBLINEAR [39]. The scikit-learn implementation also provides interoperability with other libraries popular in the Python data analysis ecosystem including: numpy [52], an array-oriented computation library; Pandas [86], a dataframe[5] manipulation library; and Matplotlib [55], a visualization library; among others.

We focus our implementations on scikit-learn as it has become a popular target for existing automated machine learning systems [41, 82, 121, 105], it provides coverage of

[2] depending on the abstractions of the underlying programming paradigm
[3] in contrast to neural-network-based approaches
[4] includes minor/major releases – as counted on GitHub
[5] a data structure used to represent (relational) tables

age	sex	height	weight	QRSduration	target
75.0	0	190.0	80.0	91.0	Supraventricular Premature Contraction
56.0	1	165.0	64.0	81.0	Sinus bradycardy
54.0	0	172.0	95.0	138.0	Right bundle branch block
55.0	0	175.0	94.0	100.0	Normal
75.0	0	190.0	80.0	88.0	Ventricular Premature Contraction (PVC)

Table 2.1: A sample of five observations with five features and the target label column for the arrhythmia dataset available via OpenML.

popular algorithms, and it has an active developer community.

2.2.2 Pipelines with Varying Performance

To illustrate the use of scikit-learn, I now present an example supervised learning task, along with four pipeline implementations.

We use the arrhythmia[6] dataset available via OpenML [116], a machine learning dataset and code repository. The task is to predict one of 16 labels associated with different types of cardiac arrhythmia. The dataset includes 279 different attributes, which were precomputed from patients' ECG recordings.[7] All attributes are of type float or boolean and there are missing values. The dataset contains 452 observations, each of which was hand-labeled by a cardiologist. Table 2.1 shows a sample of 5 observations and the first 5 features in the dataset along with the target column, which we want to predict.

Figure 2-1 shows four different pipelines that one can write using scikit-learn to solve the task at hand. All pipelines start by applying `SimpleImputer` to fill in missing values, failing to do so will raise a runtime exception with all classifiers chosen. After that, each pipeline applies a different classification algorithm. Pipeline 1 fits a linear model (logistic regression), pipeline 2 fits a decision tree, while pipelines 3 and 4 fit different types of ensemble models.

For purposes of pipeline evaluation, we use 10-fold cross-validation and compute a macro-averaged F1-score as a performance metric.

F1-score [115] is defined as

[6] `https://www.openml.org/d/5`

[7] The data was released in de-identified form

```
p1 = Pipeline([
    ("impute", SimpleImputer()),
    (
        "clf",
        LogisticRegression(
            penalty="l2",
            C=10.0,
            solver="liblinear",
            max_iter=1000,
        )
    ),
])
```

(a) Pipeline 1

```
p2 = Pipeline([
    ("impute", SimpleImputer()),
    (
        "clf",
        DecisionTreeClassifier(),
    ),
])
```

(b) Pipeline 2

```
p3 = Pipeline([
    ("impute", SimpleImputer()),
    (
        "clf",
        ExtraTreesClassifier(
            n_estimators=300,
            criterion="gini",
        ),
    ),
])
```

(c) Pipeline 3

```
p4 = Pipeline([
    ("impute", SimpleImputer()),
    (
        "clf",
        GradientBoostingClassifier(
            n_estimators=100,
            learning_rate=0.05,
        ),
    ),
])
```

(d) Pipeline 4

Figure 2-1: Four example scikit-learn pipelines for the arrhythmia dataset. All pipelines fill in missing values using `SimpleImputer` and then fit a different classifier.

$$2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

To compute its macro-average, we use the class-wise average recall and precision values when computing the formula above.

Table 2.2 presents an overview of the varying pipeline performance. Pipeline 1 and 2 have similarly low performance, despite taking different classification approaches. Pipeline 3 improves on both with a tree-based ensemble model. Pipeline 4 obtains the best performance. Note that there are also a varying number of hyperparameters which could be tuned, a subset of which may impact performance on this particular dataset. Additionally, different pipelines take different amounts of compute time when fitting to the underlying dataset (despite its small size). This diversity of performance and choices motivates the need for an automated approach to exploring the pipeline design space.

2.3 Automated Machine Learning

We now present a formulation of automated machine learning inspired by that of Feurer et al [40]. Let $\mathcal{P}$ be the set of machine learning pipelines that can be implemented using the tar-

Pipeline	Hyperparameters	F1 score	Fit time (s)
Pipeline 1	15	0.393	0.957
Pipeline 2	16	0.395	0.066
Pipeline 3	19	0.437	0.685
Pipeline 4	22	0.461	13.135

Table 2.2: The differences in predictive performance and the number of tuneable hyperparameters (despite the simplicity of the underlying pipelines) motivates the need for an automated approach to exploring the pipeline design space.

get machine learning library. Let $b \in \mathbb{R}$ be a compute time budget. Let $c : \mathcal{P} \times \textsc{Table}[D_1 \times \cdots \times D_m] \times \textsc{Vector}[D_y] \to \mathbb{R}$ be a function that returns the time to train/evaluate a pipeline. Let $\textsc{Fit}_p$, where $p \in \mathcal{P}$, be the fit function for a particular pipeline definition p.

An AutoML system optimizes the following:

$$\underset{p \in \mathcal{S}}{\operatorname{argmin}} \frac{1}{|X_{\text{test}}|} \sum_{x_i \in X_{\text{test}},\ y_i \in y_{\text{test}}} e(\textsc{Fit}_p(X_{\text{train}},\, y_{\text{train}})(x_i),\, y_i)$$
$$s.t. \sum_{p \in \mathcal{S}} c(p,\, X,\, y) \leq b,$$

where $\mathcal{S} \subset \mathcal{P}$ is the subset of pipelines evaluated by the system. The subset covered by $\mathcal{S}$ is determined by the search procedure's search space definition and exploration strategy. An automated machine learning system's candidate generation process can be viewed as a restricted form of search-based program synthesis, where programs consist of compositions of classes/functions in the target machine learning library, and the specification is a soft optimization goal given the input tabular dataset.

2.4 Software Artifacts

We refer to software artifacts as outputs (and by-products) of the broader software engineering process. This includes, but is not limited to: source code of a software project, concrete executions of programs, environment definitions, requirements documents, developer documentation, developer forums, bug reports, among others.

2.4.1 Kaggle

Two of the systems presented in this thesis leverage existing code examples. AL mines dynamic execution traces for programs that implement a machine learning pipeline, while AMS relies on a corpus of code examples to identify potential search space components and hyperparameter search spaces. For both of these, we focused the scope of programs collected (i.e. those that use machine learning in some capacity) and the scope of the datasets used by collecting programs available through Kaggle.

Kaggle [46] is a website that hosts datasets for machine learning tasks, machine learning competitions, community tutorials and programs, as well as dedicated forums. At a high-level, Kaggle is meant to provide a community space for machine learning and data analysis practitioners. As part of their offerings, Kaggle hosts users' data analysis scripts and computational notebooks.[8] Users can execute their code directly on Kaggle, which as of April 2021, allows programs to execute up to 9 hours (on 4 CPU cores with 16 GB of RAM) and have access to 20 GB of storage space.[9] Figure 2-2 provides a screen-shot of Kaggle's web-based search interface for existing notebooks.

[8]Computational notebooks are "literate computing" tools, which intersperse executable code, outputs such as visualizations or tables, and text in a cell-based environment.

[9]Kaggle also allows users limited amounts of cost-free access to GPUs and TPUs.

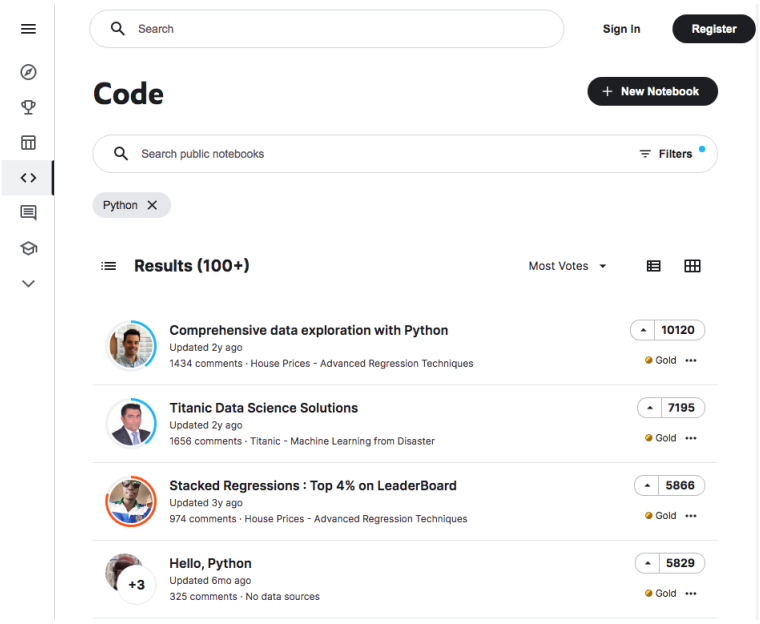


Figure 2-2: A screenshot of Kaggle's notebook search interface. Users can filter down to notebooks that are written in a particular language (Python or R), as well as rank notebooks based on characteristics such as the community's votes for notebook quality.

Chapter 3

AL

3.1 Introduction

We present[1] AL, a system that learns to generate supervised machine learning pipelines from existing supervised learning programs. AL enables analysts to automatically obtain supervised learning pipelines for a wide range problems without the need to manually select the components in the pipeline or write code that implements the pipeline. AL identifies a sequence of components (in our evaluation, these are classes in two object-oriented machine learning libraries) that are appropriate for the input data. The system then adds additional boilerplate code necessary and returns a fully executable machine learning script to the user in plain Python code.

AL's goal is to emulate the pipelines implemented in a collection of existing programs and provide a means to extend to new libraries with little to no additional development effort. By extracting the search space directly from a training set of programs, AL can automatically extend automated machine learning searches to new classes or functions used in the training set. This is a key advantage compared to existing systems that work with pre-defined subsets of libraries.

An analyst can use AL to generate new pipelines. To do so, the analyst provides AL with a dataset in tabular form. The columns correspond to features; each row is an observation. The user indicates which column of the table corresponds to the prediction

[1] A version of this chapter was previously published as a conference paper in [20].

target. AL then generates a set of pipelines consisting of method calls to API classes that process the input data, fit a learning model, and evaluate it. Each step in this pipeline takes as input the output of the previous step, and executes the API call with any remaining default hyperparameters. The system outputs a ranked list of pipelines suitable for predicting the target value of new unseen observations.

We train AL on a collection of programs, each of which implement (potentially in addition to other functionality) a supervised learning pipeline. We source our training set of supervised learning programs from Kaggle [46], a data science website that hosts competitions, tutorials, and community forums for relevant machine learning topics.

We train AL on a collection of approximately 500 supervised learning programs and their associated target datasets. AL instruments each program to collect runtime information. Specifically, the instrumentation collects a program trace, including information on the calls to the target machine learning libraries and an abstraction of the data used as parameters to those calls. The abstraction consists of a collection of feature functions that capture key characteristics (e.g. data types, summary statistics, probability densities, column correlations, missing value frequencies) of the input data to each call.

AL deploys a small set of module-level labels along with a slicing-based algorithm to extract a canonical representation of the supervised learning pipelines embedded in each dynamic trace. AL uses these canonical representations to construct a model to emulate developer-written pipelines. This model ranks new pipeline components conditioned on a partially constructed pipeline and the state of the input data presented to the new component. Given an input dataset, a pipeline depth bound, and a bound on the number of candidates per depth, AL uses the model to rank and prune candidate pipelines based on their probability, where we compute the probability of a pipeline as the product of the probabilities of each individual pipeline step, conditioned on previous steps and data state.

We compare AL to two existing automated machine learning tools: *autosklearn* [41] and *TPOT* [82]. Both systems search over a space of pre-selected API classes, tuning the selection of classes used in the pipeline and their hyperparameters. For our evaluation, we use AL to generate supervised learning pipelines for 31 data sets, none of which overlap with the data used by example programs seen by AL during training.

Our experiments show that by emulating existing pipelines, AL can produce pipelines for 17 out of 21 datasets in an average search time of 5 minutes or less, with predictive performance comparable to those produced by existing systems, with the latter given a 1 or 2 hour execution budget (see Section 3.8.3). We also show that by extracting the relevant API subset from programs, AL can eliminate the need to perform additional pre-processing of the data required by existing systems or remove the need to manually extend these systems by developers (see Section 3.8.4).

The remainder of this chapter is structured as follows. Section 3.2 walks through an example of how an analyst would use AL. Section 3.3 presents the idea of canonical supervised learning pipelines and estimating the probability of a particular pipeline. Section 3.4 presents our algorithm for extracting the component search space and the canonical pipelines embedded in our training programs' traces. Section 3.5 describes the process of collecting training programs. Section 3.6 details the use of a discriminative classifier to construct a model to predict the next component in a partially constructed pipeline. Section 3.7 presents our pipeline generation algorithm. Section 3.8 shows our experimental results. Section 3.9 describes the AL implementation and provides a reference to a demo-version repository with related code. Section 3.10 discusses threats to validity and Section 3.11 concludes.

3.2 Example

We next present an example that illustrates how AL produces a pipeline to solve a classification task for a new (unseen) dataset: the Titanic dataset [60] hosted on Kaggle. The goal is to predict which passengers survived the Titanic wreck.

3.2.1 Training AL's Pipeline Generation

We first train AL on a collection of supervised learning programs and their target datasets (in this work these programs and datasets are sourced from the Kaggle website [46]). AL collects dynamic traces from each program and extracts the canonical supervised learning pipeline embedded in each trace. It then uses these dynamic traces to build a model that predicts that probability of the next pipeline step conditioned on previous pipeline stages

and on characteristics of the data at the current pipeline stage. AL uses this model to drive the exploration of candidate pipelines and produce candidates that emulate the pipelines written by developers and observed in the training programs.

3.2.2 Pipeline Generation

The dataset for the Titanic problem takes the form of a CSV file. Each row corresponds to a passenger. The columns correspond to passenger characteristics such as age, gender, cabin, etc. Starting with this file and an identification of the column name or index indicating the target prediction column, AL splits the file into a training dataset and a held-out validation dataset. It then runs its pipeline generation algorithm, using the conditional probability model to guide the search for good candidate pipelines (see Section 3.7).

3.2.3 AL Pipeline

Figure 3-1 presents the highest ranked pipeline generated for the Titanic dataset. The pipeline consists of 3 classes from the *scikit-learn* library and a bounded loop iterator from AL's runtime library, which are sequenced using *scikit-learn*'s `Pipeline` combinator, along with boilerplate code (which is generated by filling in a pre-defined sketch) to produce an executable machine learning pipeline. A `Pipeline` instance is constructed from steps, which are defined tuples of the form name (string) and API component. Each component in the pipeline is applied in sequence to the provided inputs.

Lines 10 to 12 read the input dataset from disk and split the dataset into training and validation data. `X_train` and `y_train` correspond to the training data, and `X_val` and `y_val` correspond to the held-out validation dataset. The pipeline defines two transformations (lines 16 to 17), which are followed by fitting a logistic regression classifier (line 18). Line 22 fits each of these components in sequence, by calling `.fit` on the `Pipeline` instance. Line 25 evaluates the pipeline's performance on the held-out validation set. To do so, the `Pipeline` instance's `.score` method applies the transforms `t0` and `t1` to `X_val`, predicts the outputs using the model, and compares these to `y_val`. Line 28 re-fits the pipeline to the entire dataset for training, so that new predictions can use all information available. The `predict`

```
1  import xgboost
2  import sklearn
3  import sklearn.feature_extraction.text
4  import sklearn.linear_model.logistic
5  import sklearn.preprocessing.imputation
6  import runtime_helpers
7
8  from sklearn.pipeline import Pipeline
9
10 # read inputs and split
11 X, y = runtime_helpers.read_input('titanic.csv', 'Survived')
12 X_train, y_train, X_val, y_val = train_test_split(X, y, test_size=0.25)
13
14 # build pipeline with transforms and model
15 pipeline = Pipeline([
16   ('t0', runtime_helpers.ColumnLoop❶(sklearn.feature_extraction.text.CountVectorizer❷)),
17   ('t1', sklearn.preprocessing.imputation.Imputer❸()),
18   ('model', sklearn.linear_model.logistic.LogisticRegression❹()),
19 ])
20
21 # fit pipeline
22 pipeline.fit(X_train, y_train)
23
24 # evaluate on held-out data
25 print(pipeline.score(X_val, y_val))
26
27 # train pipeline on train + val for new predictions
28 pipeline.fit(X, y)
29
30 def predict(X_new):
31   return pipeline.predict(X_new)
```

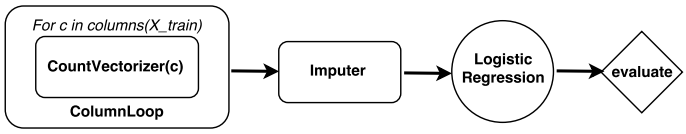


Figure 3-1: Top ranked Python program generated by AL to perform classification on the Kaggle dataset `titanic`. The diagram depicts the canonical pipeline implemented in the accompanying source code. AL populates predefined boilerplate code with the sequence of classes it chose to implement the pipeline. In this case, it chose a `CountVectorizer`❷ to convert string columns to numeric columns, and applies this in a loop using `ColumnLoop`❶. `Imputer`❸ fills in missing values, and `LogisticRegression`❹ learns to predict binary labels (i.e. survived or not).

function (lines 30 to 31) produces new predictions for new data provided by the user.

The first component of the pipeline (line 16) is a class that applies a transform that converts strings into tokens, counts the number of times each token appears in each string, and replaces the string in the data set with columns that count the number of occurrences of each token. This transformation is required because the classifier (line 18) works only with numeric data (and fails if presented with a dataset that contains text). Note that this class needs to be applied in a loop over the columns. Applying it to the table as a whole fails, as the API expects a one-dimensional vector to produce the appropriately shaped output. Removing string columns altogether would lose important information in the dataset such as gender or cabin class columns. Similarly, simply mapping each string

to a unique integer would fail to extract relevant subtokens from string columns, such as passengers' titles from the passenger name column (e.g. Miss, Mr, Master). Both of these features correlate with passenger survival [34].

The second class in the pipeline (line 17) applies a transform that imputes missing data (filling in missing data with the mean over entries present in the same column). Again, this transformation is required because the classifier fails if there is any missing data. Because this imputation class works only with numeric data, the pipeline also requires the previous application of a transform (such as the string to token count conversion transform in the example pipeline) that converts strings to numeric values. We highlight that the string to numeric conversion, and the filling in of missing values currently have to be handled manually when using other systems (see `https://github.com/EpistasisLab/tpot/blob/master/tutorials/Titanic_Kaggle.ipynb` for an example that uses this dataset with *TPOT*). The removal of these manual steps underscores the advantages of learning from example programs as employed by AL.

All of the components in the pipeline, with exception of the custom combinator `ColumnLoop` in AL's runtime library, were extracted from AL's example supervised learning programs during AL's training. In each step, the input parameter to the API consists of the output from the previous pipeline step along with default hyperparameters.

In practice, we expect that an AL user would automatically generate a set of pipelines, which they can choose to further modify by editing the pipelines' Python source code. However, even an initial pipeline can deliver strong results, with no additional developer interaction. For example, when we submitted predictions produced by the pipeline in Figure 3-1 in Fall 2017 to the corresponding Kaggle leaderboard, our predictions outperformed 91% of the submissions.

Now that the use of AL has been outlined via an example, we dive into the technical concepts underlying our contribution.

3.3 Canonical Supervised Learning Programs

Let $\mathcal{I}$ be shorthand for the input covariates, previously introduced as $\textsc{Table}[D_1 \times \cdots \times D_m]$ in Section 2.1.1. Let $\mathcal{Y}$ be shorthand for the input target vector, previously introduced as $\textsc{Vector}[D_y]$ in Section 2.1.1. We refer to a class, function, or method in an API that implements some desired functionality as a component. Let $\mathcal{T}$ be the set of data transformation components (Section 2.1.3) available in the target API. Let $\mathcal{L}$ be the set of classification/regression (Section 2.1.2) learning algorithms available in the target API and $\mathcal{H}$ be the type of a model after fitting any necessary parameters. Let $\mathcal{E}$ be the set of evaluation functions (Section 2.1.2) available in the target API.

Let $I_{\text{train}}, I_{\text{val}} \in \mathcal{I}$ and $Y_{\text{train}}, Y_{\text{val}} \in \mathcal{Y}$, where *train* refers to the training set and *val* refers to the validation set. Let $\mathcal{P}$ represent the set of component-based programs implementing a supervised learning pipeline. A program in this set takes as inputs training data $train = (I_{\text{train}}, Y_{\text{train}})$ and held-out validation data $val = (I_{\text{val}}, Y_{\text{val}})$. The program performs necessary transformations to I_{train} (and correspondingly, I_{val}), learns a model $h \in \mathcal{H}$ using *train* and evaluates the performance of the learned hypothesis on *val*. The output of the program is the trained model and the associated evaluation metric on the held-out validation data.

A program in $\mathcal{P}$ is represented as a five tuple $(train, val, T, l, e)$, where *train* is training data, *val* is held-out validation data, $T = (t_1, ..., t_k)$ is a sequence of k data transformations where $t_i \in \mathcal{T}$, $l \in \mathcal{L}$ is a supervised learning algorithm, and $e \in \mathcal{E}$ is an evaluation metric.

Figure 3-2 shows the evaluation semantics of such a program. We use I^i_{train} to denote the output of applying transformation t_i to input I^{i-1}_{train}, with $I^0_{\text{train}} = I_{\text{train}}$. The output of this sequence of transformations is used to learn a hypothesis over $I^k_{\text{train}}, Y_{\text{train}}$ using the learning algorithm l. To evaluate the hypothesis learned, we apply the same sequence of transformation components T, which may have computed parameters over I_{train} (i.e. been fitted), to I_{val} and use e to score the output of $h(I^k_{\text{val}})$ relative to Y_{val}. The output of the program is the hypothesis and its score on the held-out validation dataset.

$$\textsc{Transform}_{t_i}(\langle I_{\text{train}},I_{\text{val}}\rangle) \rightarrow \{I'_{\text{train}} \leftarrow t_i(I_{\text{train}}); \langle I'_{\text{train}},t_i(I_{\text{val}})\rangle\}$$
$$t_i(X) \rightarrow \textit{fit on } X \textit{ if unfitted and then transform else just transform}$$
$$\textsc{Learn}_l(I_{\text{train}},Y_{\text{train}}) \rightarrow \textit{model } h \textit{ on } I_{\text{train}},Y_{\text{train}} \textit{ fit using algorithm } l$$
$$\textsc{Evaluate}_e(h,I_{\text{val}},Y_{\text{val}}) \rightarrow \textit{ score trained model over validation data using } e$$

function PROGRAM(I_{train}, Y_{train}, I_{val}, Y_{val})
  $I^k_{\text{train}},I^k_{\text{val}} = \textsc{Transform}_{t_k}(...\textsc{Transform}_{t_0}(\langle I_{\text{train}},I_{\text{val}}\rangle))$
  $h = \textsc{Learn}_l(I^k_{\text{train}},Y_{\text{train}})$
  score $= \textsc{Evaluate}_e(h,I^k_{\text{val}},Y_{\text{val}})$
  return $\langle h,\text{score}\rangle$
end function

Figure 3-2: Semantics for programs in $\mathcal{P}$, the set of canonical supervised learning programs, where braces represent a program block, and a semi-colon represents operation sequencing. These programs produce a fitted hypothesis and a hypothesis score on held-out validation data.

3.3.1 Modeling Canonical Pipeline Probability

We model the conditional probability of a canonical supervised learning program $(train,val,T,l,e) \in \mathcal{P}$ as the probability that a developer would write the sequence of operations $t_1,...,t_k,l$, given the training data. This probability model can be viewed as a "language model" over pipelines, conditioned on input data [101]. This intuition captures our goal: developer emulation during pipeline generation. We remove the evaluation component from the probability calculation as the function used to score a pipeline is defined deterministically based on the prediction task to allow comparison of multiple pipelines' performance. The probability is then defined as:

$$\Pr(T,l|I_{\text{train}},Y_{\text{train}}) =$$
$$\Pr(l|T,I_{\text{train}},Y_{\text{train}}) \prod_{i=1}^{k} \Pr(t_i|T_1^{i-1},I_{\text{train}},Y_{\text{train}})$$

where T_i^j indicates the sequence $t_i...t_j$, and T_*^0 is the empty sequence.

We approximate this probability by making a Markov assumption [9] of order j about

the transformations and learning algorithm in the program. This assumption is that the ith pipeline component is a function of only the j previous calls and the input data before the ith call. By considering the input data for the ith call, rather than the initial data, we allow some additional information flow beyond the cutoff imposed by our Markov assumption. So our approximate conditional probability is now defined as

$$\approx \Pr(l|T_{k-j}^{k}, I_{\text{train}}^{k}, Y_{\text{train}}) \prod_{i=1}^{k} \Pr(t_i|T_{i-j}^{i-1}, I_{\text{train}}^{i-1}, Y_{\text{train}}). \tag{3.1}$$

3.3.2 Abstracting Inputs for Learning

We cast the problem of learning a function to estimate a pipeline's conditional probability as a supervised learning problem itself, an approach known as meta-learning [44]. We consider each transformation t_i or learning algorithm l as a label, and train a discriminative classifier to predict the appropriate label given the previous j pipeline components and the current call's input data.

To successfully estimate the conditional probability defined in Equation (3.1), we need a data abstraction that is flexible across different inputs. This flexibility must account for varying dimensions, datatypes, and underlying distributions. We define such an abstraction operation $\alpha : \mathcal{I} \cup \mathcal{Y} \to \mathbb{R}^p$ which summarizes input data using a real-valued vector of dimension p. The vector is designed to capture key characteristics of the data. We often refer to the application of α as *summarizing* the input and the values it produces as *meta-features*.

The *feature map* that defines α presents a trade-off between the richness of the representation, sparsity in training data, and computational cost. Producing very rich representations reduces the likelihood that we will observe many instances of the vector in our training data, and may be more expensive to compute during pipeline search. Very simple representations can fail to capture important features of the input data. Based on our own experience and the existing literature [3, 96], we define α using the following combination of features, which are averaged column-wise where relevant. Probability/cumulative density functions used are defined with a set of fixed distribution parameters and

Table 3.1: Comparison of data meta-features used in *autosklearn*'s meta-learning and AL's α

Meta-Feature	Autosklearn	AL
Class entropy	✓	✗
Class probability	(max, min, mean, std)	(max, min, mean)
Ratio of rows to columns	✓	✗
Kurtosis	(max, min, mean, std)	mean
Number of columns	✓	✓
Number of rows	✓	✓
Number of features by type	✓	✓
Number of classes	✓	✓
Ratio of feature types	✓	✗
Skewness	(max, min, mean, std)	mean
Symbol counts	(max, min, mean, std, sum)	(max, min, mean)
Count of rows with missing values	✓	✗
Count of features with missing values	✓	✗
Percentage of missing values	✓	✓
Cross-column correlation	✗	(max, min, mean)
Non-zero counts	✗	(mean)
PDF/CDF for common distributions	✗	(normal, chi2, exponential, gamma)
Other summary stats	✗	(geometric mean, median, zscore, IQR, std)

applied to each value in a column.

- **Type Features**: Collection type (e.g. list, array, set); distribution of types in collection elements (e.g. real, integer, string).

- **Features for Numeric Data**: Arithmetic mean; Geometric mean; Median; Minimum; Maximum; Maximum and minimum of z-score; Interquartile range; Skew; Kurtosis; Maximum and minimum of probability density function evaluation under normal, chi-squared, exponential, and gamma distributions; maximum and minimum of cumulative density function evaluation under normal, chi-squared, exponential, and gamma distributions; correlation; Count of missing values.

- **Features for Categorical Data**: Size of domain; Minimum, maximum, and mean frequency of elements in domain; Count of missing values.

Table 3.1 compares the features used in computing α with the features used by *autosklearn* in its meta-learning.

3.4 Extracting a Component Search Space

Our system eliminates the need to manually select a set of relevant API components for constructing a supervised learning pipeline. This approach relies on dynamic program traces collected by executing existing supervised machine learning programs. We focused on Python programs that contained calls to two popular data science libraries: *scikit-learn* [89], and *XGBoost* [24].

The program traces, collected through dynamic instrumentation, contain the API calls made by the program along with information on the arguments and memory locations associated with each call. We describe the details of our instrumentation in Section 3.5.1. To extract a canonical pipeline from such a trace, we need to label each call as an element in $\mathcal{T}$, the set of available data transformations, $\mathcal{L}$, the set of available learning algorithms, or $\mathcal{E}$, the set of available evaluation functions.

The sets of calls for each of the three labels are then used to train a model that predicts the probability of observing a particular component, given prior components and the state of the input data. For example, we collect all calls labeled $\mathcal{T}$ and train a model that predicts the probability of observing a given transformation component, such as `sklearn.preprocessing.Imputer`, next in the pipeline.

3.4.1 Dynamic Trace Slicing For Canonical Program Extraction

Algorithm 1 presents how we 1) label each API call in a dynamic trace, and 2) extract a canonical supervised learning program. We exploit the package hierarchy in *scikit-learn* and *XGBoost*'s bindings to label as an element of $\mathcal{T}$ any function or appropriate method call for a class defined in *scikit-learn*'s *decomposition*, *feature_extraction*, *feature_selection*, *pipeline*, *preprocessing*, and *random_projection* modules.

Similarly, we label as an element of $\mathcal{E}$ any function or appropriate method call for a class defined in *scikit-learn*'s *calibration*, *metrics*, *model_selection* modules. We also use the fact that all learning algorithms in *scikit-learn* and *XGBoost*'s Python bindings are implemented as classes with a `fit` method to apply the respective learning algorithm to data. We identify all calls to `fit` that are not already labeled and label these as elements in $\mathcal{L}$.

We say that a call that has been labeled as an element in $\mathcal{L}$ is a *seed* for a canonical supervised learning program. There may be multiple seeds in a single dynamic trace, yielding multiple canonical programs. For each seed in the trace, we slice the trace forward using the data dependencies on the seed call's return value to identify calls that depend on the output of that seed. These calls are labeled as elements of $\mathcal{E}$, if not previously labeled. We then slice the trace backwards using the data dependency information for the seed call's parameters to identify calls that are inputs to the learning algorithm. These calls are labeled as elements of $\mathcal{T}$, if not previously labeled. Concatenating the backward slice, the seed, and the forward slice produces a canonical program.

It is important to note that the traces we extract are a linearization of the original slice. So for example, if the slice contains two transforms A and B with edges into learning algorithm C, but there are no edges between A and B, the linearized trace extracted would correspond to A,B,C, where A and B are placed in an arbitrary (deterministic) order. This linearization is a requirement for our pipeline language model, which is restricted to pipelines with sequential composition.

Figure 3-3 shows an example canonical pipeline extracted from our training data. Each component is labeled according to its role in the pipeline. T stands for transform, L stands for learning algorithm, and E for evaluation.

3.5 Collecting Training Data

We collected and analyzed a set of supervised learning programs crowdsourced through Kaggle [46], a data science website that hosts machine learning competitions, datasets, tutorials, and forums. Users are provided with an extensive environment to write and execute their own scripts in popular data science languages such as Python and R, with access to Kaggle's datasets. We used Kaggle's *Meta-Kaggle* dataset [61], which includes existing user scripts. We downloaded the input data for these programs, and used this data to reproduce program executions. We learn from Python programs that made at least one call to our target libraries: *scikit-learn* and *XGBoost* [24]. Both libraries are used in existing automated machine learning tools [41, 82, 27] and were used often in Kaggle scripts.

Algorithm 1 Extract canonical supervised learning programs from a dynamic program trace. Each call is labeled as part of $\mathcal{T}$ (T), $\mathcal{L}$ (L), or $\mathcal{E}$ (E)

INPUT: D, a dynamic trace; P, a limited set of predefined mappings from API calls to labels. Let SliceFWD(d, s) and SliceBWD(d, s) respectively, compute a forward and backward slice on dynamic trace d starting from call s using data dependencies.

OUTPUT: A set of labeled canonical supervised learning programs from a single trace.

```
function Canonical-Prog(D, P)
    progs ← ∅
    Label subset of calls in D using P
                                  ▷ Seeds correspond to calls fitting a learning algorithm
    seeds ← {call ∈ D | call.label = l ∨ (call.label = None ∧ call.method = .fit)}
    for s ← seeds do
        s.label ← L
        bwd ← SliceBWD(D,s)
        Label calls in bwd as T if not previously labeled
        fwd ← SliceFWD(D,s)
        Label calls in fwd as E if not previously labeled
        prog ← Concatenate(bwd,seed,fwd)
        progs ← progs ∪ prog
    end for
    return progs
end function
```

We removed scripts that exceeded a predefined similarity ratio over token subsequences [94]. This filtering aims to remove scripts that have few semantic differences. These "duplicates" are often produced as a result of the ad-hoc version management common in data science development [63]. Our final training set consisted of approximately 500 programs from the *Meta-Kaggle* dataset. These programs targeted one of 9 tasks (meaning they perform predictions over 9 different datasets).

3.5.1 Program Instrumentation

To obtain program traces from which to learn, AL uses dynamic instrumentation. The goal of this instrumentation is to identify key execution events (particularly, assignments, calls and returns) to maintain a simplified data flow graph, which we refer to as a value graph. Nodes in the value graph are created based on execution events and annotated with data to allow trace extraction.

To effectively instrument the program, AL first performs multiple rewrites. In particular, any function (or method) call is hoisted such that it assigns to a fresh intermediate variable,

```
[('TfidfVectorizer', 'T'),
  ('TfidfVectorizer.fit', 'T'),
  ('TfidfVectorizer.transform', 'T'),
  ('TfidfVectorizer.transform', 'T'),
  ('TruncatedSVD', 'T'),
  ('TruncatedSVD.fit_transform', 'T'),
  ('TruncatedSVD.transform', 'T'),
  ('StandardScaler', 'T'),
  ('TransformerMixin.fit_transform', 'T'),
  ('StandardScaler.transform', 'T'),
  ('RandomForestClassifier', 'L'),
  ('BaseForest.fit', 'L'),
  ('ForestClassifier.predict', 'E')]
```

Figure 3-3: A canonical pipeline extracted from an existing program trace during AL's training. Each tuple consists of a call and a component label. This pipeline uses `TfidfVectorizer` to convert strings to counts and normalize them, `TruncatedSVD` to perform dimensionality reduction, `StandardScaler` to scale the reduced data, and `RandomForestClassifier` to fit a classifier.

and the corresponding call in the original statement is replaced with the intermediate variable. Then AL adds calls to its instrumentation runtime library, which maintains information, detailed below, during execution. We now detail the actions taken by the instrumentation for key events.

Assignments: When an assignment executes, AL records a new node in the value graph for each left-hand-side variable. To record data dependences, AL extracts all variables used on the right hand side of the assignment and adds a directed edge from the latest graph node associated with that variable to the new nodes just created. If there are no variables on the right-hand side (e.g. the assignment is a constant expression), no dependences are added. To identify the appropriate node for each variable, we rely on their memory address, which can be obtained in CPython by the `id` function. The instrumentation uses the memory address as a key into a hash table containing graph nodes. Finally, the newly created graph nodes are stored in the slots associated with the corresponding value of `id` for each left-hand-side-variable.

Calls: When a call executes, AL records a new node in the graph with an identifier for the call site, an identifier for the particular call, source code information and any arguments

used for the call. For each variable used in a call argument, AL adds a directed edge from the node associated with the last assignment to that variable to the new call node. The call node created also includes a summary of each call argument, produced using the output of AL's data abstraction function.

In contrast to a more general data flow analysis, we highlight that for AL we have the advantage of only caring about calls to a subset of libraries in each program (*scikit-learn* and *XGBoost*). Caring about a smaller subset reduces the calls that have to be instrumented, and given their standard usage patterns, allows us to simplify instrumentation, for example by eliding instrumentation for more complex Python constructs like generators, conditional expressions, lambdas, among others. This may result in incomplete value graphs, but we have not found this to impact AL's performance.

Returns: When a return event occurs, AL adds a new node for the return and creates an edge from the node associated with the corresponding call to the current new node. If the return value is further assigned to a variable (as is often the case, given that we hoist calls), the following *Assignment* execution event handles the remainder of the necessary value-graph updates.

Loops: Iteration in example programs can unduly affect AL's language model. For example, if a program iterates over each row of a table and applies a particular transformation t, our probability for t would be biased by the number of rows in that table. This behavior would also violate the defined semantics of transformations, which apply to a table as a whole not to a row, leading to an incorrect probability computation. To resolve this issue, AL adds a static identifier to each for and while loop in the original program, as well as a dynamic identifier assigned when a for or while loop body is entered during execution. AL then only records events in the body of the loop during the first iteration associated with the current dynamic identifier, effectively unrolling the loop once for instrumentation purposes during execution. AL keeps track of the corresponding dynamic and static identifiers for that loop, such that a new entrance in the body of the loop (e.g. a new call to a function that contains the loop in question) would once-again produce nodes for the value graph.

After a program has finished executing, we post-process the value graph to facilitate extraction of program traces. Specifically, AL extracts call nodes and merges edges as

necessary, so that our final graph contains an edge from call A to call B if the return value of A flows into B, with no intermediate function C modifying A's return value.

We encapsulated these ideas in an instrumentation library, `https://github.com/josepablocam/python-pl`, that relies on Python's built-in debugging hooks (`sys.settrace`) to carry out the instrumentation tasks outlined previously. This library uses a combination of source code and runtime type information to identify calls to functions in its own code, relevant libraries (i.e. those we want to track) and irrelevant libraries. It also disambiguates between calls made from the input program and calls that may be internal to libraries (which we do not care for, as AL focuses on public API components). The current version of AL is implemented with the same general approach, but relies more heavily on inserting calls to the runtime instrumentation library in the rewritten source code.

3.6 A Discriminative Classifier for Supervised Learning Programs

We use multi-label logistic regression with L_1 regularization, a standard machine learning algorithm, to model the conditional probability of supervised learning pipelines.

Recall that in Equation (3.1) we defined the conditional probability of a canonical supervised learning program to be approximately

$$\approx \Pr(l|T_{k-j}^{k}, I_{\text{train}}^{k}, Y_{\text{train}}) \prod_{i=1}^{k} \Pr(t_i|T_{i-j}^{i-1}, I_{\text{train}}^{i-1}, Y_{\text{train}}) \tag{3.2}$$

which has a Markov assumption of order j.

We instantiated j to 2 in our implementation, which means our model considers only the two previous pipeline operations, along with the input to the current operation, when computing the probability of the next component. In order to choose j, we performed k-fold cross validation over next-component prediction. A new set of training programs would potentially require a different j, but this new j value can be determined empirically using a similar approach.

We also abstract out the input data using our α data summarization function. We

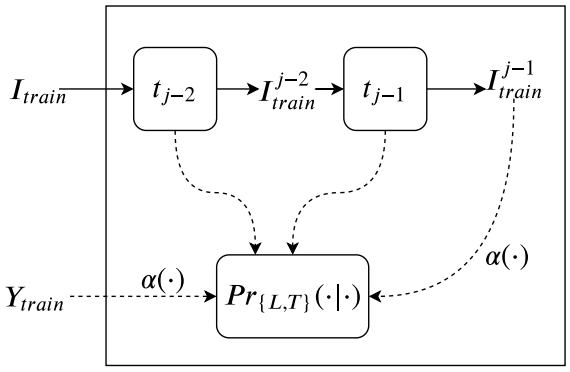


Figure 3-4: Graphical representation of our modeling of canonical pipeline next-component choice.

trained two separate classifiers: one for transformations and one for learning algorithms. Let $\Pr_T$ be the conditional probability computed by the former classifier, and $\Pr_L$ be the conditional probability computed by the latter classifier.

We then redefine the (unnormalized) conditional probability for a supervised learning program to

$$\Pr_L(l|t_{k-1},t_{k-2},\alpha(I^k_{\text{train}}),\alpha(Y_{\text{train}})) \prod_{i=1}^{k}\Pr_T(t_i|t_{i-2},t_{i-1},\alpha(I^{i-1}_{\text{train}}),\alpha(Y_{\text{train}}))$$

We summarize this model through a simple graphical representation, shown in Figure 3-4. We consider the previous two pipeline components (t_{j-2} and t_{j-1}) and a summary of the current state of the dataset (i.e. the input into the next pipeline component, $\alpha(I^{j-1}_{train})$ and $\alpha(Y_{train})$). The corresponding model then predicts the probability for the next transformation (Pr_T) or learning algorithm (Pr_L).

We trained the two classifiers on the canonical supervised learning programs extracted in Section 3.4. We do so by constructing two training corpuses: one for transformations and one for learning algorithms. For each we produce a collection of n-grams (i.e. a sequence of n preceding elements) of APIs for each call that is associated with the given role label. The target label for each observation, which is our prediction target, corresponds to the

```
<program> ::= <transform>* <learn> <score>
<transform> ::= <transform_fit> <transform_apply>
<transform_fit> ::=
   t_i = #.fit(X)
 | t_i = ColumnLoop(#).fit(X)              (for-loop)
<transform_apply> ::= X = #.transform(X)
<model> ::= m_i = #.fit(X, y)
<score> ::= m_i.score(X, y)
```

Figure 3-5: Grammar for programs generated by AL. `.fit`, `.transform`, and `.score` are part of the API for *scikit-learn* and *XGBoost* components, which we use to instantiate our set of transforms ($\mathcal{T}$), learning algorithms ($\mathcal{L}$) and evaluation ($\mathcal{E}$). Elements of the form # represent holes to be filled with API components during search. The current AL implementation uses the Scikit-Learn `Pipeline` class to simplify the code generated, but we present the equivalent step-by-step grammar here.

associated library class (e.g. `sklearn.preprocessing.Imputer`).

Recall that the goal of this model is to estimate the probability of a pipeline being written by a developer, given the input training data. From this perspective, the probability model functions as a "language model" for pipelines.

3.7 Generating Supervised Learning Programs

Now that we have a concrete way of quantifying the conditional probability of different supervised learning programs, we introduce our approach to generating new programs when given input data by the user.

3.7.1 Generation Approach

Figure 3-5 shows the grammar for the pipelines AL generates. While transformations in the pipeline are composed sequentially, bounded iteration can be expressed directly through a class in AL's runtime library. In particular, `ColumnLoop` performs a `for-loop` over the columns of the data, applying a given transformation iteratively to all columns in the input parameter.

AL enumerates pipelines in a breadth first search up to a bound on the number of components (depth) and a bound on the number of pipelines per depth. The search uses

AL's conditional probability models to rank components and greedily prune pipelines. The PredictProgramExtension function in Algorithm 2 shows how these models are used to predict the next component in a pipeline.

The Generate function in Algorithm 2 shows the overall search algorithm. At each depth, the algorithm takes the top k programs after extension. These programs are the product of adding calls in descending order of conditional probability to candidate programs, which are themselves sorted in descending order of conditional probability, and pruning the resulting set to the top k based on their current estimated probability. Each program is then extended with a learning algorithm fitting step to construct complete programs of that depth. After the final set of programs has been produced, we sort the generated programs in descending order based on the evaluation metric on a held-out validation dataset. Note that this validation set *does not* overlap with the eventual test set for the pipelines produced. Rather, it is a sampled subset of the training data. This search strategy is equivalent to beam search, with the main distinction that we add the validation-based pruning to the final set of pipelines obtained.

When given a new dataset, if the number of rows exceeds a default limit (in the current implementation: 10,000), AL executes its search over a subset of size equal to the limit, which is sampled randomly with replacement. Similarly, if the number of columns exceeds a default limit (in the current implementation: 3,000), the grammar is restricted to remove bounded column loops. These two choices were made empirically and result in adequate pipelines with reduced search times.

3.7.2 A proposal to extend AL with hyperparameter optimization

The current pipeline generation approach implemented in AL (detailed in Section 3.7) produces sequential pipelines, where each component is invoked with default hyperparameters. And while our evaluation (Section 3.8) shows that these pipelines can deliver competitive performance, extending AL with hyperparameter optimization may produce further performance gains. In this section we describe a proposal for doing so that would build on our current implementation of AL.

In particular, to incorporate hyperparameter tuning, we take inspiration from existing

Algorithm 2 Greedy Enumeration of Supervised Learning Programs

INPUT: Input training and validation data (I_{train},Y_{train},I_{val},Y_{val}); d, a bound on the depth of the programs; k, a bound on the number of programs per depth; classifiers M_t,M_l for transformations and learning algorithms, respectively.

OUTPUT: A sequence of pipelines solving the supervised learning task presented.

```
function GENERATE(I_train, Y_train, I_val, Y_val, d, k, M_t, M_l)
    P ← ()
    W ← (empty program)
    for depth ∈ 1...d do
        P_depth ← PREDICTPROGRAMEXTENSION(W,M_l,k)
        P ← APPEND(P,P_depth)
        if depth ≠ d then
            W ← PREDICTPROGRAMEXTENSION(W,M_t,k)
        end if
    end for
    return P sorted based on performance on I_val,Y_val
end function

function PREDICTPROGRAMEXTENSION(P, m, k)                    ▷
Extend an existing set of programs by one component
    P' ← ()
    for p ∈ P do
        ops ← Predict probability-ranked components for p using model m
        ps ← (p+op|o ∈ ops ∧ p+op executes without runtime exception)
        P' ← APPEND(P',ps)
    end for
    return first k probability-ranked programs in P'
end function
```

AutoML systems such as Alpine Meadow [105], Autobazaar [107], and ReinBO [109], which factor their pipeline generation processes into two distinct (but repeatedly applied) steps. First, these systems generate what they term a pipeline template or abstract pipeline. Abstract or template pipelines act as a sketch of the pipeline, fixing particular components and their composition order. To concretize the pipeline (or populate the template), the search system then proposes concrete configurations for any elements in the pipeline that remain abstract. In particular, concretization provides values for the subset of tuneable hyperparameters.

A similar approach can be directly applied to AL. Specifically, we can take the set of pipelines resulting from the standard AL implementation (see Algorithm 2 for details), and we can consider their hyperparameters to be *abstract* values. Subsequently, we can

provide the abstract pipeline, a hyperparameter search space for each component in the pipeline, the training/validation data required, and a time budget to an off-the-shelf tuning procedure. To produce the hyperparameter search spaces required we could incorporate the spaces that are mined from existing source code examples by our tool AMS (the second system presented in this thesis; see Chapter 4 for details).

Algorithm 3 A proposed extension to AL to incorporate hyperparameter optimization using a two-step approach

INPUT: Input training and validation data (I_{train},Y_{train},I_{val},Y_{val}); an instance of AL configured with appropriate beam search depth/width bounds and next-component prediction models; an instance of AMS which we can use to add hyperparameter search spaces; a tuning procedure TUNE, such as Bayesian optimization, which takes an pipeline, a search space for its hyperparameters, train/test data splits, and a time budget, which it uses to tune the pipeline based on values available in the space defined; a procedure ALLOC that allocates tuning time to a particular pipeline given the remaining time budget; an overall time budget b to tune hyperparameters.

OUTPUT: A sequence of pipelines with tuned hyperparameters.

```
1: function ExtendedGenerate(I_train, Y_train, I_val, Y_val)
2:     ▷ Generate pipelines using AL as usual
3:     P ← AL(I_train,Y_train,I_val,Y_val)
4:     P_tuned ← ()
5:     for p_i ∈ P do
6:         if b ≤ 0 then
7:             break
8:         end if
9:         ▷ Allocate tuning time to pipeline p_i
10:        bp ← Alloc(p_i,b)
11:        ▷ Retrieve mined hyperparameter search spaces for components in pipeline
12:        s_i ← AMS(Components(p_i))
13:        ▷ Run tuning procedure
14:        p'_i ← Tune(p_i,s_i,I_train,Y_train,I_val,Y_val,bp)
15:        P_tuned ← Append(P_tuned,p'_i)
16:        ▷ Update remaining time budget
17:        b ← b − bp
18:    end for
19:    return P_tuned
20: end function
```

Algorithm 3 details the proposed extension to AL. Line 3 runs the standard AL search, producing a set of pipelines with default hyperparameters. Line 10 allocates tuning time to a pipeline given that pipeline and the remaining time budget. For each pipeline, we retrieve the hyperparameter search space mined by AMS for the components used in that

pipeline (Line 12) and then we carry out the tuning process (Line 14) for the allotted time. Line 17 updates the remaining time budget for the next pipeline. Finally, we return the set of pipelines tuned. Note that we have left the choices of the concrete tuner (TUNE) and per-pipeline budget allocation (ALLOC) purposefully abstract. Different instantiations of these may yield different performance outcomes. We propose that a reasonable default tuner would be one based on Bayesian optimization, which has shown to be effective for hyperparameter tuning. For time allocation, a simple alternative may be uniform time allocation. However, adaptive time allocation, for example, based on the performance obtained by the pipeline with default hyperparameters, may yield better outcomes.

The expected results of this proposed extension are twofold. First, the overall pipeline generation process is expected to consume more time, as the system must now carry out the second step of tuning hyperparameters. Second, given an extended time budget, we also expect the tuned pipelines to be comparable to or outperform those with default hyperparameters (note that we can always return the pipeline with default hyperparameters if no tuned version improves on the original's validation performance).

3.8 Evaluation

We now present experimental results for AL.

3.8.1 Instrumentation Impact

AL relies on executing existing programs to extract the search space to be used for pipeline generation. As detailed in Section 3.5.1, execution information is collected by instrumenting the original programs, tracking library calls and characteristics of the input values used to each call. We executed our example programs with and without this instrumentation to assess the overhead imposed by AL's information collection. We found that the average ratio of instrumented to uninstrumented execution time was 1.67 (1.85 s.d.). The mean instrumented execution time was 4.38 minutes (5.33 s.d.), while the mean uninstrumented execution time was 3.36 minutes (4.34 s.d.). Figure 3-6 shows the distribution of runtime ratios across example programs. Over 80% of example programs have a runtime execution

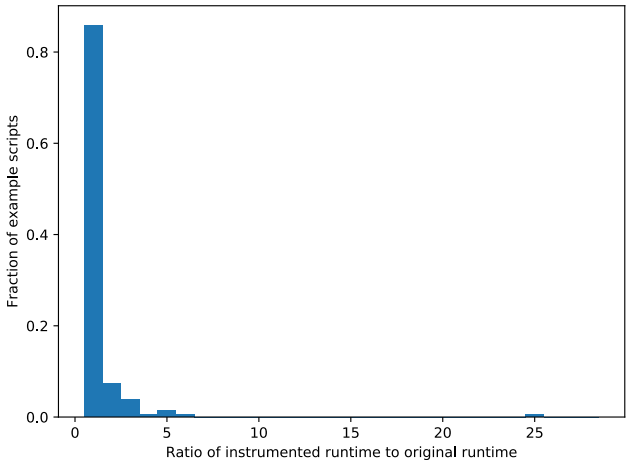


Figure 3-6: For over 80% of example programs in our Kaggle script collection, AL's instrumentation results in a runtime ratio between 1.0 - 2.0 relative to the original uninstrumented runtime

ratio between 1 and 2. There are a small number of outlier programs that have a ratio over 5 times (including one with a ratio of approximately 25.0). We found these cases to focus on programs that perform a large number of transformations on relatively large input dataframes. All our example programs were collected from Kaggle, which at the time of data collection, imposed a 20 minute execution limit [47] in their environment.

3.8.2 Benchmarking Methodology

We evaluate the pipelines produced by AL and compare against two existing systems and two baselines.

We compare AL to:

- *autosklearn* [41]: an automated machine learning system that produces pipeline ensembles using an approach termed sequential model-based algorithm configuration (SMAC) [57], which extends traditional Bayesian optimization to handle challenges core to AutoML, such as the combination of numeric and categorical hyperparameters
- *TPOT* [82]: an automated machine learning system that produces tree-based pipelines using genetic programming
- *Basic-ML*: a handwritten implementation that simulates a user with basic machine

learning knowledge. The preprocessing performed is the minimum required for all datasets to execute without program exceptions. This baseline uses `CountVectorizer` to convert string columns to token counts, replaces any missing values with zero, and applies logistic regression for classification and linear regression for regression tasks.

- *Default Prediction*: predicts the most common label for classification and the mean value for regression

We performed ten trials per benchmark dataset on AWS m4.xlarge machines with 16 GB RAM (and moved execution to m4.2xlarge with 32GB RAM when necessary for completion). For each trial we randomly split the dataset into training and test using a 75/25 split. All systems are trained/tested on the same split of the data in each trial.

We used four collections of benchmarks, totaling 31 different datasets, all of which are ***disjoint*** from the datasets used by AL's training programs.

We used 12 of the 13 datasets in the *autosklearn* paper [41]. We excluded dataset *1111*, as its size (50000 rows with 190 numeric columns and 40 categorical columns with a large domain) resulted in out-of-memory exceptions in our 32GB RAM machine and none of the AutoML systems (including AL) produced a pipeline in the time limits considered. The original datasets are available through the OpenML repository [116]. We refer to these datasets as *OpenML*.

We used all 9 datasets in the *TPOT* paper [82]. These include synthetic and real datasets. The original datasets are available through the Penn Machine Learning Benchmarks repository [84]. We refer to these datasets as *PMLB*.

For the *OpenML* and *PMLB* datasets, we convert any categorical columns to a one-hot-encoding using Pandas [79] `get_dummies` function. This is required for *autosklearn* and *TPOT* to run out-of-the-box on these datasets. When referring to column counts, we refer to the columns after one-hot-encoding.

The *OpenML* and *PMLB* datasets were chosen to compare the predictive performance of AL with other AutoML systems and baselines. We executed the AutoML systems with a budget of 1 hour per iteration, and increased this budget to 2 hours if no pipeline was produced.

The remaining 10 datasets in our evaluation were sourced from Kaggle and Mulan, a multi-label classification and multivariate regression project [113]. The Kaggle datasets in particular are reflective of the types of datasets that an analyst might encounter "in the wild". For example, we chose datasets that had a combination of different datatypes across columns and multiple prediction target columns. The former case is common in industry, where datasets are often collected and cleaned/prepared on a per-analysis basis by developers [73]. The latter case is interesting as it explores a less common task compared to single-target prediction. In particular, it exercises the flexibility of existing systems to handle input/target shapes that may have not been considered during design. For these tasks, we let AL's search run without pipeline-level timeouts. *autosklearn* and *TPOT* produce exceptions when running on these datasets out-of-the-box.

For the Kaggle and Mulan datasets, we set AL's search depth bound to 3, the greedy search bound to 30 programs per depth, and a timeout per API component call of 60 seconds. For the remaining datasets, we set AL's search depth bound to 4, the greedy search bound to 10 programs per depth, and a timeout per API component call of 5 seconds.

We used version 0.18.2 of the *scikit-learn* library and 0.9 of the *XGBoost* library for our experiments. These are the versions of the libraries used to execute the instrumented Kaggle example programs. We used version 0.2.1 of *autosklearn* and 0.9.3 of *TPOT*.

When presenting AL results, we present performance for the *top N* pipelines produced, based on AL's ranking of these programs. The ranking of programs, which takes place during AL's search, is carried out on a held-out validation portion of the training data, which is disjoint from the test set. After the pipelines have been ranked, AL re-trains the chosen pipelines on the entire training set and then evaluates on the test set. As such, a *top 1* value is directly comparable to benchmark systems, while the values listed as top 3, 5, and 10 represent the maximum score obtained by the programs within the first 3, 5, and 10 ranked programs (respectively). These scores are included to show that AL results in multiple productive pipelines.

3.8.3 Comparative Predictive Performance

Table 3.2 presents performance results on *OpenML* and *PMLB* datasets. Values in parenthesis correspond to standard deviations over dataset iterations. *autosklearn* ran with a budget of 1 hour per iteration. *TPOT* required a budget of 2 hours for datasets `179` and `184`, as no pipelines were produced with a budget of 1 hour. *TPOT* failed to produce a pipeline for `1128`, `554`, `389`, `293`, despite a 2 hour budget. *autosklearn* failed to produce a pipeline that was not a dummy classifier for `554` under both a 1 hour and 2 hour budget. AL produced pipelines with performance comparable to one of the two AutoML systems in 17 of the 21 datasets considered, despite having an average runtime of 325 seconds.

In datasets `389`, `293`, `Hill_Valley_with_noise` and `Hill_Valley_without_noise`, AL produced worse results. For `293`, a pipeline in the top 5 produces an F1 comparable to *autosklearn*. However, for the remaining datasets AL produces pipelines with lower performance than the Basic-ML baseline. After inspecting the pipelines, we identified these to be cases where AL's pipelines focused on fitting a small class of learning algorithms, rather than diversifying the candidate pipelines produced. For example, in `389` AL mainly fit decision trees or random forests, and for both `Hill_Valley_with_noise` and `Hill_Valley_without_noise` the candidate pipelines used a linear SVC or random forest. Similarly, the top 1 ranked pipelines for `293` were decision tree based, while later pipelines used a more diverse set of classifiers. Given that these pipelines produced lower performance than the default pipeline, a simple modification would be to return the default pipeline in cases where validation performance is lower for generated pipelines.

Table 3.3 presents the average runtime for AL when generating candidate pipelines for the *OpenML* and *PMLB* datasets. The search strategy results in comparable performance to existing AutoML systems, despite shorter execution times. The exception to this short runtime is dataset `554`, for which AL ran approximately 46 minutes, while both *autosklearn* and *TPOT* failed to produce a pipeline in 2 hours.

Table 3.2: AL produced a comparable pipeline to *autosklearn* or *TPOT* in 17 of the 21 datasets considered, despite having an average runtime of approximately 5 minutes compared to one or two hour budgets. In the four cases where performance was worse, we identified a lack of diversity in classifiers as an underlying cause. Values in parentheses correspond to standard deviations over split iterations.

Dataset	Top 1	Top 3	Top 5	Top 10	Autosklearn	TPOT	Basic-ML	Default Prediction	Metric	Source	Rows	Cols
1049	0.73 (0.05)	0.74 (0.05)	0.74 (0.05)	0.76 (0.04)	0.75 (0.04)	0.75 (0.04)	0.71 (0.08)	0.47 (0.00)	F1	OpenML	1458	37
1120	0.85 (0.01)	0.85 (0.00)	0.85 (0.00)	0.85 (0.00)	0.87 (0.00)	0.87 (0.01)	0.76 (0.00)	0.39 (0.00)	F1	OpenML	19020	10
1128	0.94 (0.01)	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)	0.96 (0.01)	-	0.94 (0.01)	0.44 (0.00)	F1	OpenML	1545	10935
179	0.78 (0.00)	0.78 (0.00)	0.78 (0.00)	0.78 (0.00)	0.80 (0.00)	0.79 (0.01)	0.43 (0.00)	0.43 (0.00)	F1	OpenML	48842	121
184	0.78 (0.01)	0.78 (0.01)	0.78 (0.01)	0.78 (0.01)	0.84 (0.02)	0.78 (0.02)	0.34 (0.01)	0.02 (0.00)	F1	OpenML	28056	48
293	0.78 (0.00)	0.90 (0.00)	0.90 (0.00)	0.90 (0.00)	0.93 (0.00)	-	0.76 (0.00)	0.34 (0.00)	F1	OpenML	581012	54
38	0.94 (0.01)	0.94 (0.01)	0.94 (0.01)	0.94 (0.01)	0.93 (0.02)	0.95 (0.02)	0.82 (0.02)	0.48 (0.00)	F1	OpenML	3772	52
389	0.54 (0.02)	0.56 (0.02)	0.57 (0.02)	0.57 (0.02)	0.74 (0.02)	-	0.78 (0.02)	0.02 (0.00)	F1	OpenML	2463	2000
46	0.96 (0.01)	0.96 (0.01)	0.96 (0.01)	0.96 (0.01)	0.96 (0.00)	0.96 (0.01)	0.91 (0.01)	0.23 (0.00)	F1	OpenML	3190	287
554	0.95 (0.00)	0.95 (0.00)	0.95 (0.00)	0.95 (0.00)	-	-	0.91 (0.00)	0.02 (0.00)	F1	OpenML	70000	784
772	0.50 (0.02)	0.52 (0.02)	0.52 (0.01)	0.53 (0.01)	0.51 (0.02)	0.50 (0.02)	0.42 (0.01)	0.35 (0.01)	F1	OpenML	2178	3
917	0.88 (0.02)	0.88 (0.02)	0.88 (0.02)	0.88 (0.02)	0.88 (0.03)	0.90 (0.02)	0.64 (0.03)	0.36 (0.01)	F1	OpenML	1000	25
Hill_Valley_with_noise	0.78 (0.06)	0.78 (0.06)	0.78 (0.06)	0.78 (0.06)	0.99 (0.01)	0.99 (0.01)	0.96 (0.01)	0.32 (0.01)	F1	PMLB	1212	100
Hill_Valley_without_noise	0.85 (0.03)	0.85 (0.03)	0.85 (0.03)	0.85 (0.03)	1.00 (0.00)	1.00 (0.00)	0.99 (0.01)	0.32 (0.01)	F1	PMLB	1212	100
breast-cancer-wisconsin	0.96 (0.02)	0.97 (0.01)	0.97 (0.01)	0.97 (0.01)	0.96 (0.02)	0.96 (0.01)	0.96 (0.01)	0.39 (0.01)	F1	PMLB	569	30
car-evaluation	0.97 (0.01)	0.98 (0.01)	0.98 (0.01)	0.98 (0.01)	0.97 (0.02)	0.97 (0.01)	0.73 (0.03)	0.21 (0.00)	F1	PMLB	1728	21
glass	0.64 (0.11)	0.68 (0.08)	0.68 (0.07)	0.68 (0.07)	0.65 (0.10)	0.70 (0.10)	0.47 (0.06)	0.10 (0.02)	F1	PMLB	205	9
ionosphere	0.92 (0.04)	0.93 (0.03)	0.93 (0.03)	0.94 (0.02)	0.95 (0.03)	0.95 (0.03)	0.87 (0.04)	0.39 (0.02)	F1	PMLB	351	34
spambase	0.94 (0.01)	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)	0.92 (0.01)	0.38 (0.00)	F1	PMLB	4601	57
wine-quality-red	0.34 (0.05)	0.37 (0.04)	0.37 (0.04)	0.39 (0.04)	0.36 (0.06)	0.37 (0.04)	0.26 (0.03)	0.10 (0.01)	F1	PMLB	1599	11
wine-quality-white	0.45 (0.02)	0.45 (0.01)	0.45 (0.01)	0.46 (0.01)	0.45 (0.02)	0.47 (0.03)	0.24 (0.02)	0.10 (0.00)	F1	PMLB	4893	11

Table 3.3: AL search times for *OpenML* and *PMLB* datasets.

Dataset	Seconds (s.d.)
1049	53.62 (3.34)
1120	236.88 (6.56)
1128	812.48 (28.71)
179	799.69 (77.29)
184	442.69 (12.48)
293	215.25 (8.06)
38	66.28 (3.45)
389	373.92 (17.17)
46	268.31 (11.83)
554	2774.48 (32.58)
772	28.31 (1.57)
917	43.56 (2.79)
Hill_Valley_with_noise	123.88 (2.91)
Hill_Valley_without_noise	133.61 (3.23)
breast-cancer-wisconsin	36.93 (1.71)
car-evaluation	45.84 (1.25)
glass	23.61 (1.65)
ionosphere	31.03 (1.38)
spambase	162.54 (2.61)
wine-quality-red	44.23 (3.88)
wine-quality-white	111.03 (10.02)

3.8.4 Learned Search Space

The Kaggle data has columns of different datatypes, such as string columns. For example, the `spooky-author-identification` dataset contains two columns with text: the first column is a string identifier, and the second column contains free-form text. *autosklearn* and *TPOT* fail to run out-of-the-box on these datasets because none of the pre-selected components available to *autosklearn*'s or *TPOT*'s search implement an effective transform to convert

Table 3.4: The search space extracted from example programs, paired with a simple execution strategy, allows AL to handle a broader range of datasets.

Dataset	Top 1	Top 3	Top 5	Top 10	Basic-ML	Default Prediction	Metric	Source	Rows	Cols
detecting-insults-in-social-commentary	0.77 (0.01)	0.78 (0.01)	0.78 (0.01)	0.78 (0.01)	0.78 (0.01)	0.43 (0.00)	F1	Kaggle	3947	2
housing-prices	0.84 (0.04)	0.85 (0.04)	0.85 (0.03)	0.85 (0.03)	0.80 (0.09)	-0.00 (0.00)	R^2	Kaggle	1460	80
mercedes-benz	0.52 (0.06)	0.53 (0.05)	0.53 (0.05)	0.53 (0.05)	0.51 (0.06)	-0.00 (0.00)	R^2	Kaggle	4209	377
sentiment-analysis-on-movie-reviews	0.33 (0.00)	0.43 (0.00)	0.43 (0.00)	0.43 (0.00)	0.22 (0.00)	0.14 (0.00)	F1	Kaggle	156060	3
spooky-author-identification	0.84 (0.01)	0.84 (0.01)	0.84 (0.01)	0.84 (0.01)	0.81 (0.00)	0.19 (0.00)	F1	Kaggle	19579	2
titanic	0.81 (0.02)	0.82 (0.03)	0.83 (0.03)	0.83 (0.03)	0.83 (0.02)	0.38 (0.01)	F1	Kaggle	891	11
enb	0.98 (0.00)	0.98 (0.00)	0.98 (0.00)	0.98 (0.00)	0.90 (0.01)	-0.01 (0.01)	R^2	Mulan	768	8
jura	0.63 (0.20)	0.71 (0.05)	0.73 (0.05)	0.73 (0.05)	0.65 (0.07)	-0.01 (0.01)	R^2	Mulan	359	15
sf1	-0.01 (0.13)	0.05 (0.05)	0.05 (0.05)	0.05 (0.05)	-0.02 (0.05)	-0.01 (0.01)	R^2	Mulan	323	10
sf2	0.13 (0.06)	0.14 (0.05)	0.14 (0.05)	0.14 (0.05)	0.04 (0.08)	-0.01 (0.01)	R^2	Mulan	1066	10

strings to a numeric vector. For other systems to execute, the analyst would have to manually write code that transforms strings to numeric encodings, combines dense and spare matrix representations, and then performs any remaining necessary transformations, such as imputation. AL's key advantage is it removes the need for this additional user-written code.

AL handles datasets with text fields without manually adding pre-processing because the example programs used to train AL have components that process text fields automatically. Specifically, the set of transforms extracted from its training programs includes a TF-IDF transformation [102] and a transform to convert strings to simple token frequency.

Both *autosklearn* and *TPOT* fail out-of-the-box on the Mulan datasets because neither system as currently implemented handles multivariate regression [6, 112]. An analyst intending to use these systems would have to manually piece together a separate pipeline for each target variable, making sure that transformations in the pre-processing stage are unified across pipelines after obtaining the resulting pipeline from each tool's search. AL executes without additional extension.

AL handles multivariate regression without manual extension because some of the example programs used to train AL have components that support multivariate regression and AL is able to construct successful pipelines using these components. By not introducing additional constraints on the search space or the characteristics of expected inputs/outputs, AL is able to produce executable pipelines for these datasets.

For 8 of these 10 datasets AL's top 10 pipelines include a program that outperforms the Basic-ML baseline. For 2 of these 10 datasets the performance is roughly equal. Recall that the Basic-ML baseline involves manually written code to preprocess inputs and assemble the prediction pipeline. In all cases, AL outperforms the *Default Prediction* baseline.

Performance on two of the multi-target regression datasets (*sf1* and *sf2*), although better than *Basic-ML* and *Default Prediction*, is still poor. We inspected the pipelines produced and found that the component choices were reasonable. For example, for *sf1* the top ranked pipeline applies (from the `sklearn.preprocessing` package) `LabelEncoder` in a loop over columns to convert single string values to integers, then normalizes column values by applying `MinMaxScaler`, and finally fits a ridge regression. The poor performance is reflective of the inherent difficulty of the predictive task in both datasets. The target vectors consist of the counts of three different types of solar flares in a 24 hour period. Note that AL's performance improves slightly for *sf2*, which has "much more error correction applied to it, and has consequently been treated as more reliable" [1].

3.8.5 Conditional Probability Model and Search Space Impact

AL's pipeline generation depends on the components extracted from training programs and the conditional probability model built from sequences of these components. To better understand the impact of this conditional probability model, we compared the pipelines generated by AL on the *OpenML* and *PMLB* datasets to two alternative strategies. *Sklearn-XGBoost-Random*: randomly searches over all classes in *scikit-learn* and *XGBoost*. *AL-Random*: searches only over classes extracted by AL but ranks components randomly and prunes pipelines randomly.

Table 3.5 shows the average F1 score for the top 10 pipelines generated by *AL-Random* is higher than *Sklearn-XGBoost-Random* for 19 of the 21 datasets. This highlights the effectiveness of the extracted component space. AL, which uses the conditional probability model, produces further improvements in 19 of the 21 datasets over *AL-Random*, showing the benefits of a guiding pipeline generation through pipeline probability.

3.8.6 Search Times for Different Models

Table 3.6 shows the average search time on OpenML datasets across different model configurations relative to the search time when using the *AL* conditional probability model. *Just Data* and *Just Code* show the ratios when using a conditional probability model that

Table 3.5: Using components extracted from program traces by AL during training (*AL-Random*) improves average F1 scores for the top 10 pipelines relative to a random search over the entire *scikit-learn* and *XGBoost* libraries in 19 of the 21 datasets. Adding the conditional probability model (*AL*) during search produces additional improvements in 19 of the 21 *OpenML* and *PMLB* datasets.

Dataset	AL	AL-Random	Sklearn-XGBoost-Random
1049	0.74 (0.04)	0.68 (0.13)	0.64 (0.11)
1120	0.85 (0.00)	0.82 (0.03)	0.70 (0.14)
1128	0.90 (0.12)	0.91 (0.10)	0.94 (0.02)
179	0.65 (0.18)	0.76 (0.02)	0.67 (0.13)
184	0.67 (0.06)	0.65 (0.15)	0.33 (0.23)
293	0.71 (0.28)	0.33 (0.36)	0.24 (0.39)
38	0.89 (0.03)	0.68 (0.25)	0.35 (0.37)
389	0.37 (0.17)	0.36 (0.18)	0.15 (0.21)
46	0.95 (0.01)	0.92 (0.11)	0.84 (0.18)
554	0.94 (0.03)	0.89 (0.06)	0.89 (0.05)
772	0.50 (0.03)	0.48 (0.06)	0.45 (0.09)
917	0.81 (0.06)	0.78 (0.09)	0.65 (0.15)
Hill_Valley_with_noise	0.72 (0.06)	0.63 (0.15)	0.58 (0.18)
Hill_Valley_without_noise	0.72 (0.10)	0.69 (0.19)	0.60 (0.23)
breast-cancer-wisconsin	0.96 (0.02)	0.93 (0.10)	0.85 (0.18)
car-evaluation	0.89 (0.08)	0.85 (0.11)	0.66 (0.20)
glass	0.66 (0.08)	0.61 (0.12)	0.45 (0.18)
ionosphere	0.92 (0.03)	0.90 (0.05)	0.84 (0.13)
spambase	0.94 (0.01)	0.89 (0.15)	0.82 (0.20)
wine-quality-red	0.35 (0.05)	0.33 (0.05)	0.22 (0.11)
wine-quality-white	0.44 (0.05)	0.37 (0.09)	0.28 (0.13)

conditions just on the data summarization features (output of α) or just on the previous k pipeline components, respectively. We found that for 17 of the 21 *OpenML* and *PMLB* datasets using the full conditional model resulted in a faster search time than just using data. However, when using just code, the benefit is less apparent, with the full conditional model providing faster search time in 11 of the 21 datasets. On average, *Just Data* resulted in a search time ratio of 1.38 (s.d. 0.50) and *Just Code* resulted in a search time ratio of 1.22 (s.d. 0.53), relative to the full conditional model.

3.8.7 Pipeline Distribution

Figure 3-7 summarizes the classes used in the top 1 and top 10 ranked pipelines generated by AL across benchmark datasets. Figures 3-7a and 3-7b show classes that implement a learning algorithm on the x axis and the fraction of pipelines that used this component as their learning algorithm on the y axis. A taller bar indicates a more popular learning algorithm. The top ranked pipelines have a more diverse set of learning algorithms, while lower ranked pipelines tend to default to use a random forest classifier.

Figures 3-7c and 3-7d shows similar data for transformation classes. The generated pipelines tend to scale values before applying a learning algorithm. Lower ranked pipelines

Table 3.6: Search time under different conditional probability models relative to search time under the AL model. A ratio above 1 indicates longer search time relative to AL.

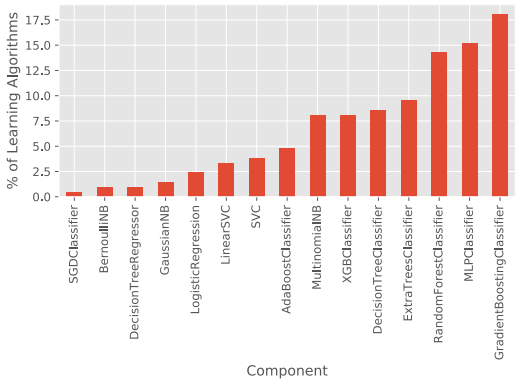
Dataset	Just Data	Just Code
1049	2.29	0.97
1120	1.13	1.60
1128	1.33	0.97
179	0.82	0.86
184	2.36	2.56
293	2.66	0.79
38	1.85	1.49
389	1.43	0.37
46	1.16	1.57
554	1.49	0.74
772	0.92	1.83
917	1.01	1.08
Hill_Valley_with_noise	1.38	0.69
Hill_Valley_without_noise	1.27	0.59
breast-cancer-wisconsin	1.13	0.96
car-evaluation	0.98	1.77
glass	0.92	1.60
ionosphere	1.19	1.23
spambase	1.29	0.78
wine-quality-red	1.23	1.75
wine-quality-white	1.18	1.44

show a larger diversity of data transformations.

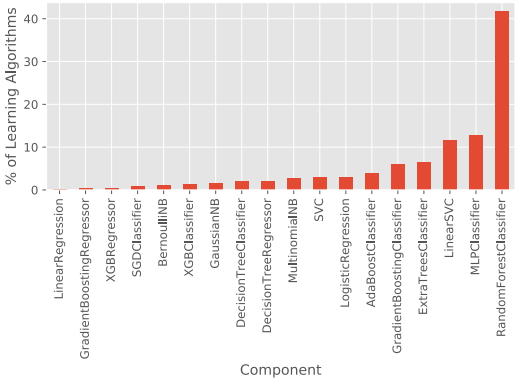
Approximately 8% of the top 10 pipelines generated contained 4 classes, 27% contained 3 classes, 41% contained 2 classes, and 25% contained a single class. The total number of pipelines possible, given the depth bounds, are on the order of 1e6 (of which on the order of 1e5 are executable without runtime errors). AL's search procedure explores a fraction (on the order of 1e3) of the possible pipelines.

3.8.8 Comparing to Kaggle User Programs

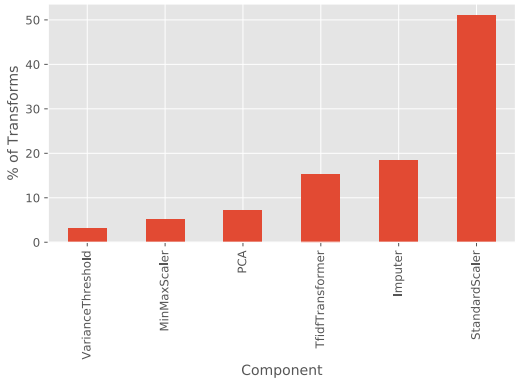
Table 3.7 shows the performance for the top ranked pipeline produced by AL for three Kaggle datasets with open submissions as of Fall 2017/Spring 2018. The first column shows the dataset. The second column shows AL's submission score, the top user score and the percentile associated with AL's score. AL outperformed 29%, 51% and 91% of submissions in the *housing-prices*, *spooky-author-identification*, and *titanic* dataset competitions, respectively, at that time.



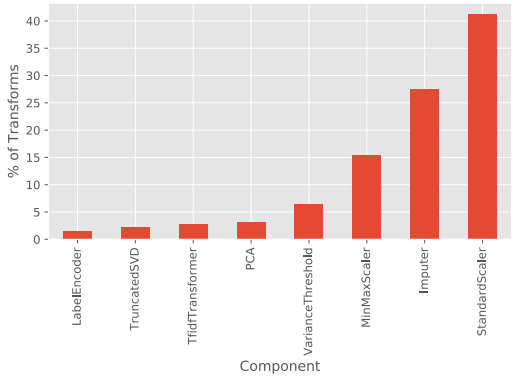
(a) Distribution of classes implementing regression/classification algorithms in top 1 generated pipelines.



(b) Distribution of classes implementing regression/classification algorithms in top 10 generated pipelines



(c) Distribution of classes implementing data transformations in top 1 generated pipelines



(d) Distribution of classes implementing data transformations in top 10 generated pipelines

Figure 3-7: Distribution of learning algorithm and transform components in the top one and top ten generated pipelines for our evaluation datasets. 8% of the top 10 pipelines generated contained 4 classes, 27% contained 3 classes, 41% contained 2 classes, and 25% contained a single class (the learning algorithm). Our algorithm produces a bounded number of programs of each depth up to a depth bound.

Table 3.7: Submissions to open Kaggle datasets. The second column should be read as AL's score, followed by the top user scored, and the percentile associated with AL's score. AL outperformed 29%, 51% and 91% of existing submissions at the time of submission (Fall 2017/Spring 2018).

Dataset	AL vs Top User	Lower/Higher Better?
housing-prices	0.16 / 0.00 (29.08 percentile)	Lower
spooky-author-identification	0.47 / 0.13 (51.55 percentile)	Lower
titanic	0.81 / 1.00 (91.51 percentile)	Higher

3.9 Implementation

We implemented AL in approximately 5000 lines of Python code. We integrated the machine learning libraries described in our evaluation (*scikit-learn* and *XGBoost*). A demo version of AL, as a Python tool, can be found at `https://github.com/josepablocam/AL-public`.

3.10 Threats to Validity

AL's ability to produce pipelines that successfully process a broad range of data depends on the quality of the example supervised machine learning programs used to train AL's conditional probability model. In particular, AL's success is dependent on having a set of programs that use relevant portions of the library API at least once (and ideally multiple times). To the extent that AL is provided with poor or very few training examples, the generated pipelines may not successfully generalize to unseen data. This risk can be mitigated by providing multiple high quality example programs.

As in all machine learning applications, it is possible that evaluation on the test set produces poor results relative to the training set. AL splits the input data into a training and held-out validation dataset to use during pipeline generation to mitigate this risk.

We compared against two AutoML tools. We believe these systems are representative of two popular types of search strategies in AutoML: model-based and evolutionary. Both of these tools choose library classes to include in their pipelines and tune hyperparameters. AL only focuses on choosing classes and performs calls with default hyperparameters. It is possible that other tools may produce higher performance pipelines on different datasets. We do not claim that AL can outperform all AutoML tools, but rather that it demonstrates the effectiveness of a novel contribution: extracting the search space of pipeline components from example supervised machine learning programs. We showed comparable performance on a collection of datasets from varied sources and also showed that learning a search space enables AL to handle additional datasets without the need to manually extend the space.

While AL identifies library components from example programs and uses these during search, there are pre-existing decisions that are made manually in the current implemen-

tation. In particular, AL is implemented to use *scikit-learn* and *XGBoost* libraries in its pipelines. We leveraged this constraint to simplify instrumentation, pipeline fitting, and code generation. Both AutoML systems we compared to are also tailored to this same set of libraries but in addition the designers manually chose the specific API components to incorporate into their search spaces.

Our implementation of AL also introduces a custom combinator during pipeline generation as part of its runtime library: `ColumnLoop`. This combinator is used to apply transformations in a bounded loop over an input value's columns. `ColumnLoop` is no longer necessary with newer versions of *scikit-learn*, which introduced `sklearn.compose.ColumnTransformer` to perform a similar task.

3.11 Conclusion

We presented AL, a new system that processes an existing corpus of machine learning programs to learn how to generate effective pipelines for solving supervised machine learning problems. AL extracts canonical pipelines from program traces and uses these pipelines to define the search space of components. It trains a conditional pipeline probability model over the extracted sequences of library classes and uses this model to guide the generation of new pipelines. Our results highlight the effectiveness of this technique in leveraging existing programs to learn how to generate supervised learning pipelines that work well on a range of problems.

Chapter 4

AMS

4.1 Introduction

We introduce[1] the use of a *weak pipeline specification* as a way to provide partial user preferences to an AutoML tool via its search space. A weak pipeline specification consists of an unordered set of API components that the end user may want to appear in the resulting pipeline. This specification can be automatically extended to produce a *strong pipeline specification* that captures additional API components of interest, defines a set of hyperparameters and values to search over, and a search procedure to sample candidate pipelines. The strengthened pipeline specification can then influence the output pipeline produced by the AutoML tool by constraining its search space.

For example, the user might provide the scikit-learn component { LogisticRegression } as a weak specification. Strengthening this specification could add other linear models (e.g. linear SVM), would specify different types of regularization (e.g. L1/L2) and their weights, and would include the search procedure (e.g. genetic programming) used to sample pipelines. This proposed model of interaction allows the end user greater control over the eventual output pipeline, without negating the key advantage of AutoML: the user need not be an ML expert.

We introduce AMS (Figure 4-1), a system that automatically strengthens AutoML search space specifications. To carry out this strengthening, AMS exploits information in a

[1] A version of this chapter was previously published as a conference paper in [19].

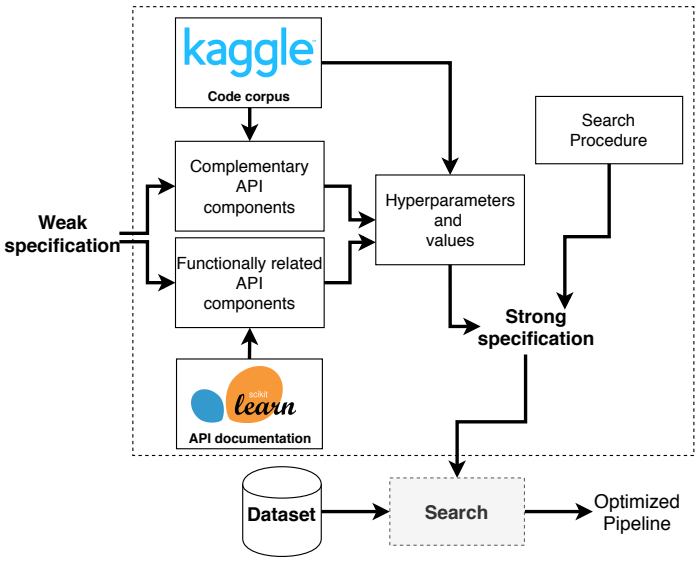


Figure 4-1: AMS system diagram. System boundaries are depicted as a dashed line. The user provides a weak specification, which is automatically extended by AMS to include complementary, functionally related API components, and key hyperparameters and a set of possible values.

code corpus and the target API's documentation. First, AMS automatically mines pairs of complementary API components from the selected code corpus, where two components are complementary if they co-occur frequently. To formalize this mining procedure, AMS uses normalized pointwise mutual information [15] to rigorously characterize co-occurrence in probabilistic terms. These mined associations can then be used to extend the initial weak specification. Next, AMS identifies unspecified API components that may be functionally related to those in the original specification. To reason about component similarity, AMS applies BM25 [97], a popular and effective measure of lexical similarity, over the API's documentation. With this metric, AMS can identify components with the highest degree of relation to those in the original specification. Next, given that machine learning pipelines are known to exhibit different performance based on hyperparameter values [91], AMS uses frequency distributions, estimated from the selected code corpus, to define a hyperparameter search space for each component in the extended specification. Finally, AMS pairs this component configuration with a search procedure, which is used to sample candidate pipelines from the given space.

We empirically evaluate AMS's predictive performance, in terms of macro-averaged

F1 score, over 9 datasets and 15 weak pipeline specifications. Our results show that, with two different search procedures, AMS produces pipelines that outperform the pipelines obtained using the initial weak specification and an expert-annotated version of the weak specification including hyperparameters and values. To quantify the extent of outperformance, we use the concept of a *win*. A pipeline wins when it obtains the highest score on a specification/dataset combination, and satisfies a minimum predictive score difference to rule out comparable scores.

When using genetic programming as a search procedure, AMS's specifications result in 38 wins compared to 12 under the weak specification extended with an expert hyperparameter space. When using random search as a search procedure, AMS's specifications result in 41 wins compared to 14 wins under the weak specification extended with an expert hyperparameter space. We also find that the pipelines produced using AMS's specification qualitatively reflect the influence of the weak specification.

In the following sections we introduce the notion of pipeline specifications (Section 4.2), provide an illustrative scenario for the use of AMS (Section 4.3), detail the approach and design of AMS (Section 4.4), present experimental results (Section 4.5), outline possible threats (Section 4.6) and conclude (Section 4.7).

4.2 Pipeline Specifications

The current usage model for AutoML typically emphasizes the lack of user involvement [124]. Under this model, the user presents the tool with their target dataset, for which they want to learn a classification pipeline, sets some computational budget, runs the tool, and accepts the pipeline produced by the AutoML tool. In this context, the AutoML tool receives no user feedback (beyond the input dataset), and the user is unable to influence the pipelines considered by the search procedure. Without any formal user feedback, the AutoML tool is unable to 1) exploit any user domain knowledge or 2) provide a pipeline that satisfies any desired user constraints (e.g. interpretability).

We propose the use of *weak specifications* as a way for AutoML users to influence the pipelines produced by automatically subsetting the relevant set of API components, thus

constraining the search space for candidates generated by the AutoML tool.

Definition 1 *Weak Specification. A weak specification is an (unordered) set of API components, at least one of which is a regressor (if performing a regression task) or a classifier (if performing a classification task).*

By providing a set of API components a user provides (partial) information regarding what they want: specifically a set of algorithms (e.g. classifiers, preprocessors) that should be considered for pipeline generation. We call this type of specification *weak* as it is incomplete along four key dimensions:

1. it does not specify what hyperparameters are relevant
2. it does not specify what values hyperparameters can take on
3. other relevant API components may be missing
4. it does not specify any order or compositional operators used to generate new pipelines from these components

Providing a weak specification allows a user to exert influence on the final pipeline produced, while at the same time not requiring deep API or machine learning expertise, as they do not have to manually detail the complete space. For example, a user can enforce a degree of interpretability on the optimized pipeline by writing a specification with a single linear model (e.g. logistic regression).

Definition 2 *Strong Specification. Let h_c be a map from a subset of hyperparameters for component c to a collection of possible values. Let C be a map from component i to its respective h_i. Let P be a search procedure to generate candidate pipelines. A strong specification is a triple of the form $\langle C,(h_1,...,h_n),P\rangle$.*

A strong specification, in effect, defines a search space for an AutoML tool. We propose that this space can be derived from the weak specification, which expresses (partial) user preferences. In the following sections, we detail our approach to doing so. But first, we introduce an illustrative scenario to demonstrate a use case for AMS.

4.3 Illustrative Scenario

We follow the journey of a forensic scientist who is not a machine learning expert but wants to classify glass fragments [37, 26]. The forensic scientist has a high level understanding of different learning and preprocessing algorithms but is not aware of the various hyperparameters, possible values, or other suitable algorithms to consider. The scientist has heard of AutoML and thinks this might be a suitable tool to explore pipelines. However, they have clear constraints: no tree-based ensemble models, as the pipelines need to be easily interpretable. Unfortunately, AutoML tools are known to often produce tree-based ensemble models [50, 38], which are challenging to interpret [51].

They spent some time on the internet and found a related tutorial that detailed a scikit-learn [89] pipeline that may work for their use case (case 1 in Table 4.1).

The scientist will use this example pipeline (without specifying any hyperparameters that can be tuned or values they can take on) as a weak specification. To evaluate their progress, they will use an existing classification dataset, "glass" [37], consisting of continuous measurements for 7 types of glass. The scientist performs a random 80/20 split for training/testing and evaluates pipelines using macro-averaged F1 score.

The scientist starts by naively running their specification directly as a pipeline with default hyperparameters, which results in an initial F1 score of *0.43*. Next the scientist uses the specification components, with default hyperparameters, as a configuration for the AutoML tool TPOT [82], which uses genetic programming to generate candidate pipelines. Applying TPOT to the weak specification (with no hyperparameters defined in the search space) results in a better score of *0.51*.

After consulting with a machine learning colleague, the scientist sets up a defined hyperparameter space (i.e. which hyperparameters to tune and set of possible values) for each component in the specification. The scientist then applies the same genetic programming search to the new configuration, resulting in pipeline number 3 in Table 4.1. Note that the shape of the optimized pipeline is the same as in the prior step, but now the regularization penalty and its weight varies. This step raised their score to 0.57.

The scientist now goes back to the original weak specification and uses AMS to auto-

matically strengthen this weak specification (rather than manually specifying the full space). AMS extends the weak specification using a code corpus and the API's documentation. Applying the same search procedure to AMS's specification now results in the highest score of *0.75*. The final pipeline retains polynomial features, but replaces the variance threshold selector with a selector based on a specified false positive rate. The pipeline then stacks a SGDClassifier (with hinge loss) and uses logistic regression with an L1 penalty (to produce sparse coefficients). This embodies the spirit of the initially given specification, but substantially outperforms the rest of the approaches.

Table 4.1: Summary of scenario iterations based on the "glass" dataset showing the progression of score improvements. Note that component names are abbreviated for brevity.

#	**Pipeline**	**Description**	**Score**
1	PolyFeatures, MinMaxScaler, VarianceThreshold, LogisticRegression	Initial (naive) weak specification as a pipeline with default hyperparameters.	0.43
2	StackingEstimator(LogisticRegression), LogisticRegression	Applying AutoML tool TPOT (Genetic Programming) to the original specification without defining any hyperparameters.	0.51
3	StackEstimator(LogisticRegression ([Penalty: L1, Cost: 10]), LogisticRegression	Same as #2, but with expert-defined hyperparameter space for regularization (cost) and penalty.	0.57
4	PolyFeatures, SelectFPR, StackingEstimator(SGDClassifier [Loss: Hinge]), LogisticRegression [Penalty: L1, Cost: 100]	Applying genetic programming to the strong specification generated by our approach (AMS) given the weak specification.	0.75

4.4 AMS

We introduce AMS, a system that automatically strengthens weak pipeline specifications using an existing code corpus, an API's documentation, and a plug-in search procedure. Figure 4-1 shows a diagram of the system. AMS takes the user's weak specification as input. The system first extends the set of API components considered in the specification. To perform this extension, AMS relies on a code corpus, which exercises the target API, and

on the API's natural language documentation. After the specification has been extended, AMS uses the code corpus to identify key hyperparameters for the API components in the specification and includes sets of possible values they can take on. AMS then pairs this set of component configurations with a search procedure to produce a strong specification. The search procedure can then be used to iteratively sample and evaluate candidate pipelines, resulting in a final optimized pipeline.

We now present details on each of these steps.

4.4.1 Unspecified (but Useful) API Components

AMS first extends the initial specification with additional components, which the user may not have included. Given a specification S and a new component c, c may be added to S if it satisfies one of the following two conditions: c is commonly used with a component already in S, or c could replace a component already in S.

The goal of the first condition is to identify *complementary components*. For example, if a classifier is often used with a particular preprocessing step, we say these components are complementary. The goal of the second condition is to identify *functionally related* components, which are alternatives to each other. For example, two different linear classifiers would be considered functionally related.

AMS relies on two different sources of information to identify components that satisfy each of these conditions. We first address complementary components.

Complementary Components

To identify complementary components, AMS exploits information from a crowd-sourced corpus of scripts, which exercise the target API. Each script in the corpus was written to target a single dataset, therefore two components used in the same script may be complementary. By using a code corpus to identify such components, AMS can automatically produce and update its inventory of complementary components to reflect current ML practices.

From the code corpus, the system extracts all scripts that contain a call to our target API library and (statically) records the set of API components used[2] in each script. The

[2]these uses may included calls that are not realized at runtime, but are observed in the original source

intuition is that these sets can be used to measure the likelihood of components co-occurring, and that complementary components must (by definition) co-occur more frequently.

Formally, we compute the normalized pointwise mutual information (NPMI) [15] over the collection of all (unordered) pairs of co-occurring API calls in our code corpus to identify complementary components. Let X and Y be two random variables, representing possible components, defined over the the domain of our target API library. We define NPMI for two components $x \in X$ and $y \in Y$ as

$$\text{NPMI}(x,y) = \frac{\log_2\left(\frac{p(x,y)}{p(x)p(y)}\right)}{-\log_2(p(x,y))} \tag{4.1}$$

where $p(x)$ is the fraction of pairs where either element is x divided by the number of all pairs, similarly for $p(y)$, and $p(x,y)$ is the fraction of pairs (x,y) or (y,x) divided by the number of all pairs.

NPMI ranges between -1 and 1, where -1 means the components never co-occur, 1 means the components always co-occur, and 0 means the components are independent. We compute the NPMI over the set of all pairs of co-occurring components (i.e. API components called in the same script). Eliminating pairs with an NPMI less than or equal to zero yields pairs of varying degree of complementarity.

When given a weak specification, we can identify all NPMI-positive pairs that share a component with the specification. For each such pair, the new potential component corresponds to the element in the pair that is not in the original specification. If more than one component in the original specification supports (i.e. co-occurs with) a new component, we compute an average NPMI. For each possible new component, we compute a weighted sum of the average NPMI and the fraction of original specification components that support it. The weighted sum balances average NPMI and support fraction based on a user-defined weight $\alpha \in [0,1]$. We then take the top K_{comp} new components and add them as complementary components to the original specification. Algorithm 4 describes this procedure.

---

code

Algorithm 4 Extracting Complementary Components

INPUT: A collection P of pairs of API components co-occurring in a code corpus; a function NPMI that computes the normalized pointwise mutual information of two API components; a specification $S = \{c_1,...,c_n\}$; a weight $\alpha \in (0,1)$ to combine NPMI and support fraction; and an integer K_{comp} for the maximum number of complementary components to take.

OUTPUT: A new specification S' extended with at most K_{comp} new components.

```
procedure ComplementaryComponents
    ▷ Map from co-occurring pair to accumulator list of NPMI scores
    npmis ← {}
    for c ∈ S,(p1,p2) ∈ P do
        if (c ∈ (p1,p2)) ∧ (p1 ∉ S ∨ p2 ∉ S) then
            new ← p2 if c1 = p1 else p2
            ▷ Accumulate the npmi score
            npmis[new] ← npmis[new] :: NPMI(p1,p2)
        end if
    end for
    ▷ Compute average npmi and support fraction
    ▷ Combine using α to create score
    scores ← {}
    n ← LEN(S)
    for comp ∈ npmis do
        vals ← npmis[comp]
        scores[comp] ← AVG(vals) * α + (LEN(vals)/n) * (1 − α)
    end for
    S' ← S ∪ GETTOPK(scores,K_comp)
    return S'
end procedure
```

Functionally Related Components

The goal of identifying functionally related components is to include algorithm alternatives in the specification. For example, the user's weak specification may indicate that they are interested in using linear models, but they may have not exhaustively listed all linear model alternatives. This task raises the challenge of reasoning about the semantics of API components. Rather than reason about component semantics, we rely on a simpler notion of similarity.

We would like to define a function $\text{SIM}(c_1, c_2)$ that computes a score for two API components, c_1 and c_2, such that a higher score corresponds to higher degree of semantic similarity. Given a component c_i, we can then sort all possible components in our target API in descending order based on their similarity score with respect to c_i.

AMS exploits the fact that the target library has natural language documentation for

each component (as part of its developer documentation), which we assume details key aspects about their functional behavior. By mining the API's documentation, AMS can be used to automatically identify functionally related components in new target libraries or new versions of previously used libraries without the need for extensive expert annotation.

We define $\text{SIM}(c_1,c_2)$ to be computed over the documentation[3] for c_1 and c_2 and instantiate it to a classical relevance/similarity scoring technique: BM25 [97]. BM25, detailed below, produces a score for a document, given a query and a corpus of documents. A higher score indicates a higher degree of lexical correlation between the document and the query.

$$\text{BM25}(D,Q) = \sum_{i}^{n} \text{IDF}(C,q_i) \frac{f(q_i,D)*(k_1+1)}{f(q_i,D)+k_1*\left(1-b+b*\frac{\text{Len}(D)}{\text{AvgLen}(C)}\right)} \tag{4.2}$$

where D is a document, $Q=(q_1,...,q_n)$ is a query comprised of q_i terms, C is a corpus of documents, and k_1 and b are score hyperparameters[4].

In our setting, the documentation for an existing component in the weak specification corresponds to the query, a particular API component's documentation corresponds to the document, and the entirety of the API's documentation corresponds to the document corpus.

AMS uses SIM to retrieve, and append, the top K_{rel} new components for each component in the original weak specification (i.e., we do not consider any complementary components added for purposes of this procedure). Algorithm 5 describes this procedure.[5]

4.4.2 Identifying Hyperparameters and Values

Machine learning practitioners often spend a significant amount of time not just choosing pipeline components, but also tuning the hyperparameters associated with each component. Performance can significantly increase by identifying the appropriate hyperparameter

[3]We perform standard preprocessing of the documentation strings such as tokenization, stemming, and extension with the path of the given component in the library's module structure.

[4]We use the gensim [95] BM25 implementation, where $k_1=1.5$ and $b=0.75$ are implementation-defined constants.

[5]AMS also exposes functionality to label a weak specification component as "include" (by appending `:1`) or "exclude" (by appending `:0`), indicating that it must be included or excluded from the strengthened specification, respectively.

Algorithm 5 Extracting Functionally Related Components

INPUT: A collection C of API components; a map M from API component to documentation; a function SIM that computes the BM25 score between a query string and a document; a specification $S = \{c_1, ..., c_n\}$; and an integer K_{rel} for the maximum number of functionally related components to take per component in S.

OUTPUT: A new specification S' extended with at most K_{rel} functionally related components per component in the original specification.

```
procedure FunctionallyRelatedComponents
    ▷ Set of empty API components
    extension ← ∅
    for c ∈ S do
        scored ← {(c', SIM(M[c], M[c'])) for c' ∈ C if c' ∉ S}
        c_K ← GetTopK(scored, K_rel)
        extension ← extension ∪ c_K
    end for
    S' ← S ∪ extension
    return S'
end procedure
```

values for a given dataset and pipeline [91].

AMS relies on the corpus of scripts that make calls to the target API to identify the set of relevant hyperparameters and possible values. This design choice hypothesizes that an AutoML system should focus on tuning the set of hyperparameters and hyperparameter values that human developers focus on tuning.

For each script in our code corpus that imports the target API, we parse the source code and identify calls to API class constructors. We extract the set of optional arguments in each constructor call and record each pair of (argument name, argument value) as a hyperparameter setting. The value recorded corresponds to a constant in the constructor call, or points to an unknown placeholder.

When given a specification, AMS takes each API component and identifies the set of top K_{params} hyperparameter names observed in the mined code for that component, along with the top K_{vals} values observed for each of the names. AMS adds the default value for each hyperparameter to the set of possible values (obtained by introspecting the class definition), and then emits this as the corresponding hyperparameter search space. Algorithm 6 describes this procedure.

Figure 4-2 shows a specification, originally just `sklearn.linear_model.LogisticRegression`

Algorithm 6 Adding API Component Hyperparameters and Values

INPUT: A map P from API components to hyperparameter names and frequencies observed in calls; a map V from hyperparameters to values and their frequencies observed in calls; a specification $S = \{c_1, ..., c_n\}$; an integer K_{params} for the maximum number of hyperparameters to consider per component; and an integer K_{vals} for the maximum number of values per hyperparameter to consider.

OUTPUT: A new specification S' with at most K_{params} hyperparameters per component and at most $K_{\text{vals}}+1$ (including default value) per hyperparameter.

```
procedure HyperParamsAndValues
    ▷ Empty map from component to hyperparameter space
    S' ← {}
    for c ∈ S do
        params ← GetTopK(P[c], K_params)
        ▷ Empty configuration for component c
        c_config ← {}
        for p ∈ params do
            values ← GetTopK(V[p], K_vals)
            ▷ Append default value, if not included
            values ← values ∪ GetDefaultValue(p)
            c_config[p] ← values
        end for
        S'[c] ← c_config
    end for
    return S'
end procedure
```

, extended with a complementary component (Algorithm 4), a functionally related component (Algorithm 5), and hyperparameters and values (Algorithm 6).

4.4.3 Search Procedure

To fully satisfy the definition of a strong specification, AMS must add in a specific search procedure to the extended specification. In particular, the current implementation of AMS outputs a dictionary-structured search space definition that is compatible for use with TPOT [82], a genetic programming-based AutoML tool. AMS' codebase also includes a random search procedure to produce length-bounded sequential pipelines.

Genetic Programming

We use TPOT [82], a genetic programming based AutoML tool, as a search procedure. When using TPOT, we use the search space defined by AMS as the configuration dictionary

```
{
  'sklearn.linear_model.LogisticRegression': {
    'C': [100000.0, 7, 1.0],
    'penalty': ['l1', 'l2']
  },
  'sklearn.feature_extraction.text.TfidfTransformer': {},
  'sklearn.linear_model.SGDClassifier': {
    'loss': ['log', 'hinge'],
    'penalty': ['l2', 'elasticnet']
  }
}
```

Figure 4-2: A weak specification extended with one complementary component, one functionally related components, and two hyperparameters/values per component (plus the implementation-defined default value, if not already covered).

provided to TPOT's optimization process.

Random Search

AMS's implementation includes a hierarchical random search procedure to generate sequential (i.e. API components are chained in sequence) pipelines. Random search is known to perform better for algorithm configuration than equally simple alternatives such as grid search [12] and has also been successfully applied to related software engineering areas such as product line configuration [81]. To generate a pipeline, the search module samples a depth (up to a bound), then for each step in the pipeline it samples an API component from the configuration specified. For each hyperparameter in the chosen component's configuration, the search samples a value and sets it in that component's constructor. The search distinguishes between preprocessing and classifier components to generate valid candidate pipelines (i.e. the last step must always be a classifier). Candidate pipelines are cached to avoid re-training/evaluating pipelines, however, there is no effort to exhaustively search the space and if a pipeline is re-sampled a given number of times (100 in our implementation), the search procedure terminates.

4.5 Evaluation

We now present our experimental results, which evaluate individual parts of our system (RQ1-RQ3, RQ5) and the overall performance of AMS (RQ4). First, we characterize the complementary API components extracted from our code corpus (RQ1). We evaluate AMS' ability to retrieve functionally related API components (RQ2). We then characterize the use of hyperparameters and their values in our code corpus, and evaluate the possibilities for improving classifier performance based on this information (RQ3). We evaluate AMS's ability to produce specifications that result in higher performance (RQ4). And finally, we explore the impact of the code corpus size on AMS's mined hyperparameters and complementary components (RQ5).

For our evaluation, we implemented AMS and its evaluation in approximately 4400 lines of Python. We use scikit-learn [89], a popular Python machine learning library, as the target API for pipelines. To mine complementary components and identify hyperparameters/values, we use the meta-Kaggle [61] dataset as our code corpus. The meta-Kaggle dataset contains over 3300 Python scripts.

4.5.1 RQ1: Complementary API Components

AMS mined 285 normalized PMI (NPMI) positive association pairs from our code corpus. These associations cover 69 different components (39.2% of all components in scikit-learn).

Table 4.2 details the distribution of associations based on the algorithmic role of each of the components, along with their mean and standard deviation NPMI.

To evaluate the effectiveness of these NPMI-based component extensions, we conduct the following experiment. We take each script in our code corpus, and extract the set of scikit-learn components used. Using 10-fold cross validation (CV), we split this collection of components into a training fold and test fold. We use each training fold to compute NPMI, and we use the corresponding test fold to evaluate the associations. For each set (ground truth) in the test fold, we take each component individually and use it as a query term to retrieve the top $K_{\text{comp}} \in [1,5]$ complementary components based on our approach (Algorithm 4, with $\alpha = 0.5$). We then compute precision as the fraction of retrieved

Table 4.2: NPMI-based association rules mined from our code corpus to identify complementary API components categorized by algorithmic role. When both components in the association have the same role, we elide one for brevity.

Rule Type	# Rules	Mean Norm. PMI	SD Norm. PMI
classifier	78	0.18	0.11
(classifier, cluster)	1	0.37	-
(classifier, decomposition)	3	0.16	0.13
(classifier, feature extraction/selection)	29	0.19	0.15
(classifier, preprocessor)	31	0.20	0.13
(cluster, decomposition)	2	0.45	0.26
(cluster, preprocessor)	1	0.27	-
(cluster, regressor)	3	0.10	0.05
(decomposition, feature extraction/selection)	7	0.17	0.19
(decomposition, preprocessor)	4	0.24	0.26
(decomposition, regressor)	3	0.20	0.25
feature extraction/selection	3	0.35	0.25
(feature extraction/selection, preprocessor)	10	0.25	0.20
(feature extraction/selection, regressor)	4	0.17	0.12
preprocessor	3	0.32	0.26
(preprocessor, regressor)	6	0.20	0.21
regressor	97	0.19	0.07

components that are present in the full ground truth component set. Note that recall is not an appropriate measure of performance for evaluating complementary components, as recall implies our extensions need to be complete, but by definition we will only be able to cover components with strong co-occurrence patterns. Given this, we focus on precision.

We found that 82.68% of the sets in the test folds were covered (i.e. we were able to identify at least one complementary component). For $K_{\text{comp}} = 1$, we found that our NPMI-based approach yields a precision of 60%. This precision declines as expected when we increase K_{comp}, with a precision of approximately 28% when $K_{\text{comp}} = 5$. Based on these results, we configured AMS to use $K_{\text{comp}} \leq 3$. Figure 4-3 summarizes these results. These levels of complementary component retrieval sufficed for improved performance on our end-to-end benchmarks, but further improving complementary component precision could deliver additional gains.

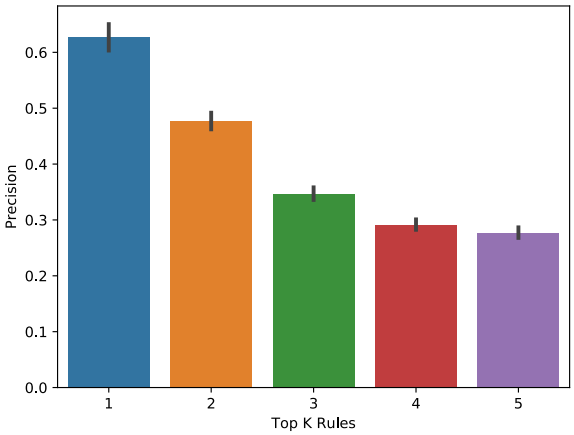


Figure 4-3: NPMI-based component extension can produce at least one complementary component for 82.68% of our test observations, with precision of approximately 60% when $K_{\text{comp}} = 1$.

4.5.2 RQ2: Functionally Related API Components

To evaluate AMS's retrieval of functionally related components, we manually annotated our BM25-based ranking of API components for a given query component. To determine if two components were functionally related, we outlined a set of conditions that they should satisfy. Given a specification component Q (for query) and a possible extension component R (for related), we mark them as functionally related if they satisfy the following:

- R could replace Q in a pipeline without raising an exception for the same dataset.
- Q and R belong to the same class of operators (e.g. classifier, regressor, value normalizer, decomposition algorithm, loss function).
- If Q/R are classifiers/regressors, they must respect output shape constraints: a multi-task model can replace a single task model, but not vice-versa.
- If Q is (non-)linear, R must be always (non-)linear or must be (non-)linear based on a hyperparameter (e.g. SVM with a linear kernel)
- If Q is ensemble-based, R must be ensemble-based with one exception: R can be non-ensemble based if it is related (based on these rules) to the weak model class ensembled in Q.

- If Q is not ensemble-based, R may be ensemble-based if it uses a weak model class related to Q to create its ensemble.

To carry out our experiment, we randomly sampled 50 classes from scikit-learn and used these as queries. We chose to sample 50 classes as this covers approximately 28% of the components available in scikit-learn and balanced the need for detailed manual annotation. For each query, we retrieved the top 10 API components based on: 1) our BM25 metric, 2) cosine similarity using averaged pre-trained neural embeddings (which have been shown to be effective for the related task of code search [18]), and 3) a uniform random metric. We used (2) to compare the use of BM25 with another unsupervised approach to semantic similarity. We used BERT embeddings derived from a scientific text corpus [10]. We used (3) as a baseline to control for the extent to which our target API (scikit-learn) may have redundant components resulting in functionally related results through chance.

Figure 4-4 presents our results. The BM25-based ranking performed comparably (with no statistically significant difference) to the embeddings based approach. A random ranking results in approximately 10% functionally related results, across the top 1, 5, and 10 query results. In contrast, BM25 results in close to 72%, 55%, and 44% functionally related results across the same cutoffs, respectively. We opted to use BM25 in AMS, in contrast to the neural embeddings approach, given their comparable performance and the added advantage of avoiding the additional storage requirements imposed by per-token embeddings.

Note that while for purposes of this experiment, we allow functionally related components to include ensembled-variants of non-ensemble models, in our tool implementation users can exclude ensembles through a simple command line flag.

While we evaluated functionally-related API component retrieval using BM25 and cosine-similarity using BERT embeddings, AMS can use other information retrieval metrics. We also note that the task of specification strengthening, in the context of AutoML, is related but not the same as pure information retrieval. In particular, we generate a *new* search space configuration based on a weak specification, rather than searching over a stored (and pre-enumerated) set of configurations.

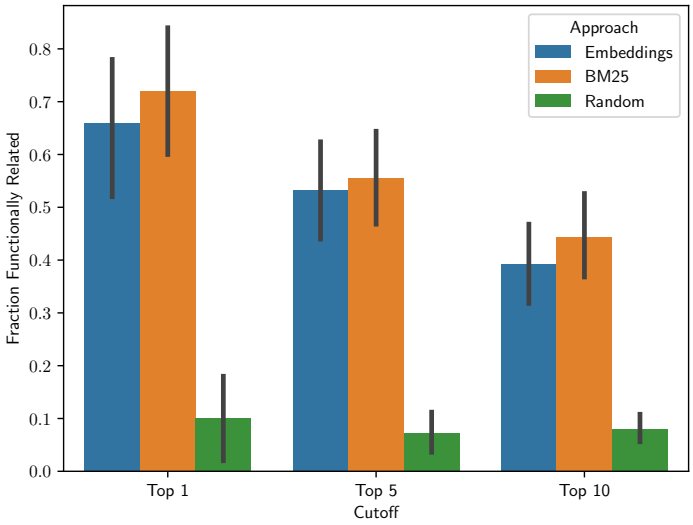


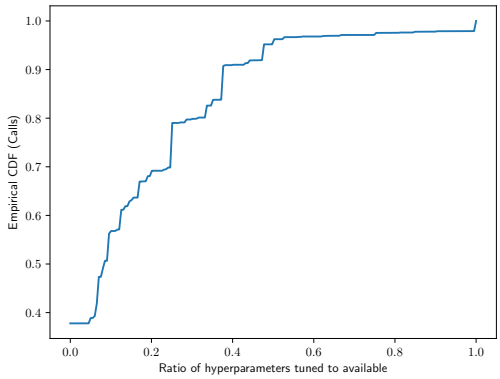
Figure 4-4: For 50 randomly sampled query API components, BM25 can retrieve close to 72%, 55%, and 44% functionally related components based on top 1, 5, and 10 cutoffs.

4.5.3 RQ3: Hyperparameters and Values

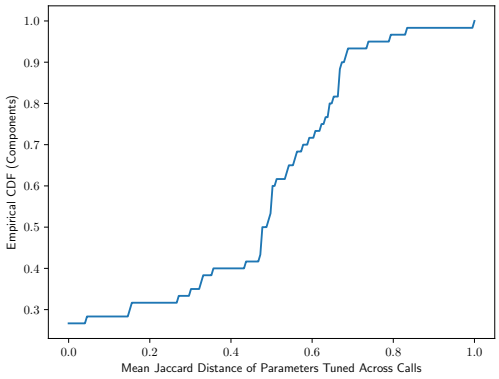
Figure 4-5 characterizes the hyperparameter tuning observed in our code corpus. In particular, we found that over 50% of the calls tune (i.e. explicitly set a value in the call) for under 20% of the hyperparameters available (4-5a); for about a third of API components the set of hyperparameters tuned is similar across calls (4-5b) as computed using Jaccard similarity over the hyperparameter sets; and for over 70% of the hyperparameters observed, user calls choose few values (under 10 distinct values) (4-5c). This aligns with our intuition that human developers tend to tune a small set of hyperparameters, these are consistent across datasets/pipelines, and there are popular values that developers choose for each.

To demonstrate the possible impact of hyperparameter tuning, we performed the following experiment. We collected five datasets from the Penn Machine Learning Benchmarks (PMLB) [84]. The five datasets are healthcare-related classification tasks. We collected these 5 dataset to be independent from those used in RQ4. We then identified the top 5 most common classifiers[6] from our code corpus. For each classifier, we extracted the top 3 hyperparameters and top 3 values for each hyperparameter, along with the default

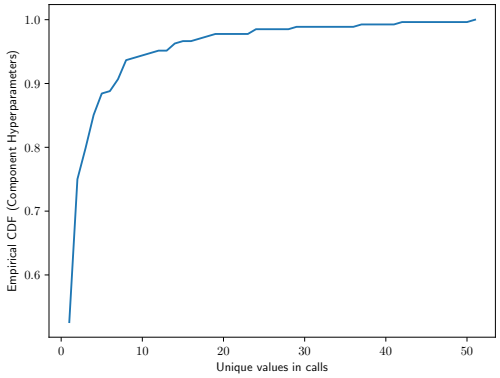
[6] excluding SVM, which did not terminate within a reasonable computing budget without additional data pre-processing for these datasets



(a) Fraction of hyperparameters tuned



(b) Distance between sets of hyperparameters



(c) Unique hyperparameter values used

Figure 4-5: Characterizing hyperparameter tuning in our code corpus.

values. We performed grid search over these values to evaluate all possible configurations. We then compared the best macro-averaged F1 score [43] from the grid search with the score obtained under the default configuration.

Figure 4-6 shows our results. In almost all cases, the hyperparameter space defined by the code examples in our corpus contained a setting which would have improved performance with respect to the default configuration. For the ensemble-based classifiers, ExtraTreesClassifier and RandomForestClassifier, this improvement could have been up to 10% on two of the datasets.

4.5.4 RQ4: Performance of Strong Specifications

Our performance experiments compare the following approaches:

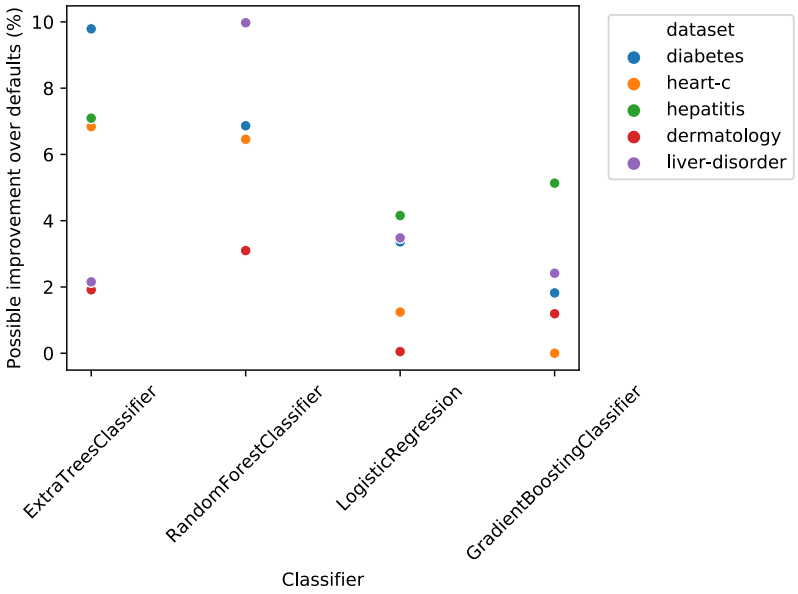


Figure 4-6: Possible improvements in macro-averaged F1 score by using hyperparameter settings in our code corpus, compared to the performance using defaults.

- Weak Spec.: runs an ordered version of the original weak specification as a pipeline directly.
- Weak Spec. + Search: carries out a specified search procedure over the components defined in the weak specification (with default hyperparameters).
- Expert + Search: uses the set of hyperparameters/values defined in TPOT's default classifier configuration [83] for each component in the specification, and applies the specified search procedure. This choice of hyperparameter space corresponds to an expert AutoML developer identifying key hyperparameters and values. We also evaluated writing our own hyperparameter space and found that it performed comparably or worse, so we elide for brevity.
- AMS + Search: applies AMS to the weak specification to produce a full search space and then applies the specified search procedure.

For these experiments, we consider both search procedures available to AMS: genetic programming (using TPOT) and random search.

Table 4.3 presents the individual components used to create the weak specifications for our experiments. We chose components that covered common machine learning operations:

Table 4.3: Components used to produce weak specification. *Expert + Search* uses TPOT's pre-defined hyperparameter search space [83] for each component.

Short name	Component
lr	Logistic Regression
rf	Random Forest
dt	Decision Tree
scale	Min-max value scaling
poly	Extract polynomial features
var	Variance-based feature selection
pca	PCA decomposition

value scaling, feature derivation, feature selection, dataset decomposition, and varied forms of classification.

We produced 15 weak specifications by combining the following 5 pre-processing weak specifications with each of the three classifiers (lr, rf, dt) - as outlined in Table 4.3: {} (no-preprocessing), {scale}, {poly, scale}, {poly, scale, var}, and {poly, scale, pca, var}.

For our experiments we used all classification datasets from the original TPOT paper [82]; 9 in total. These datasets are: Hill-Valley-with-Noise, Hill-Valley-Without-Noise, breast-cancer-wisconsin, car-evaluation, glass, ionosphere, spambase, wine-quality-red, and wine-quality-white. All datasets are available through PMLB [84].

Our experiments used macro-averaged F1 score as a performance metric, where a higher score corresponds to better performance. Each search procedure uses this same score metric in their internal search loop. For each benchmark, dataset, and search procedure combination, we carried out 5-fold cross-validation (CV) with each of the approaches outlined previously. In each CV iteration, the training fold is used to find an optimized pipeline, and the test fold is used for evaluation. All approaches were provided a budget of 5 minutes per CV iteration (i.e. 25 minutes per dataset, for each specification and approach combination).

We evaluate AMS with the following configuration: a weak specification can be extended with at most 3 complementary components ($K_{\text{comp}}=3$), where the npmi/support fraction weighing parameter is set to 0.5 ($\alpha=0.5$), for each specification component we include up to 4 functionally related components ($K_{\text{rel}}=4$), and we tune the top 3 hyperparameters per component ($K_{\text{params}}=3$) by choosing from the 3 most common values per hyperparameter ($K_{\text{vals}}=3$). We set the depth bound for the random search procedure to 4.

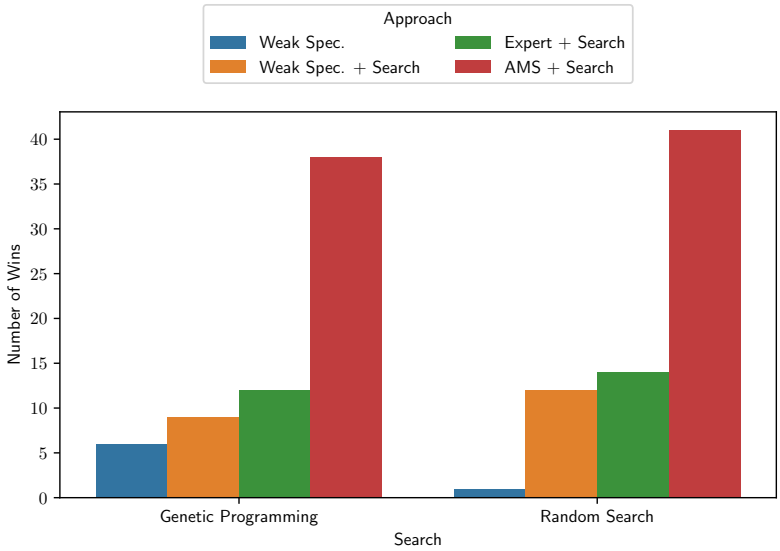


Figure 4-7: Wins for each approach across 270 experiments. Within a given search procedure, an approach *wins* when it obtains the highest average 5-fold CV test-fold performance for a dataset and weak specification combination, *and* this score is at least 1% higher (in absolute terms) than the next best score.

Figure 4-7 presents a count of the wins for each approach across both search procedures [23]. An approach *wins* when the average of the 5-fold CV test-fold performance metric is the highest across approaches for a given dataset and weak specification combination, *and* the score is at least 1% (in absolute terms) higher than the next best score. We introduced a minimum performance difference threshold to eliminate cases where multiple approaches perform roughly equally on a specification/dataset combination. We varied the minimum difference threshold from 1% to 5% (absolute) and found that AMS obtained more wins than other approaches in all cases.

When using genetic programming as a search procedure, we see that *Weak Spec.* obtained 6 wins compared to 9 wins for *Weak Spec. + Search*. Under random search, *Weak Spec.* obtained 1 win and *Weak Spec. + Search* obtained 12 wins. *Expert + Search* obtained 12 wins when using genetic programming, and 14 wins when using random search. Under both search procedures, using AMS produced the majority of wins: 38 in the genetic programming experiments and 41 in the random search experiments.

Figure 4-8 presents the distribution of the top-10 scikit-learn operators as a fraction

of the total count of operators in pipelines produced by genetic programming using AMS's strengthened specification for two different weak specifications.

Comparison to other program-mining based AutoML tools We also compared AMS to AL [20]. AL mines dynamic program traces to learn a probabilistic model for ML pipelines and uses this to generate sequential pipelines. A key advantage of AMS is that users can strengthen specifications without the need to collect a new corpus that reflects their initial specification. AMS can also mine information from otherwise un-executable programs and without access to the programs' target datasets, while AL requires program execution for its dynamic analysis.

To compare AL and AMS, we consider the weak specification of scikit-learn components [7]:

```
{LogisticRegression, LinearSVC, StandardScaler}
```

and run experiments on our 9 datasets. We use 5-fold CV, pair pipelines between CV folds in order to appropriately perform comparisons after removing pipelines that do not qualitatively reflect the weak specification, and then compute wins on the paired pipelines. If the pipeline for a system does not reflect the specification, the other system's pipeline is assigned the win. AL is trained on the corpus presented in [20], which is restricted to programs it has already executed and from which it has extracted dynamic traces.

When AL is trained on the subset of programs that use at least one weak specification component, and AMS mines this same set of programs, we find that AL can produce pipelines that *still* deviate from the weak specification (as the full program traces may contain additional components). 21 of the 45 pipelines generated by AL did not reflect the weak spec, while all of AMS do. After removing these pipelines, AMS obtains 29 wins and AL obtains 9. When AMS is trained on the full AMS corpus, AMS's wins increase to 35 (and all pipelines continue to reflect the specification) and AL's decrease to 4. Finally, when AMS is trained on the AMS corpus and AL is trained on the full AL corpus (without any specification-related program pruning), 26 of the 45 AL pipelines do not reflect the weak specification. After removing these pipelines, AMS obtains 42 wins and AL obtains 1 win.

[7] names abbreviated for brevity

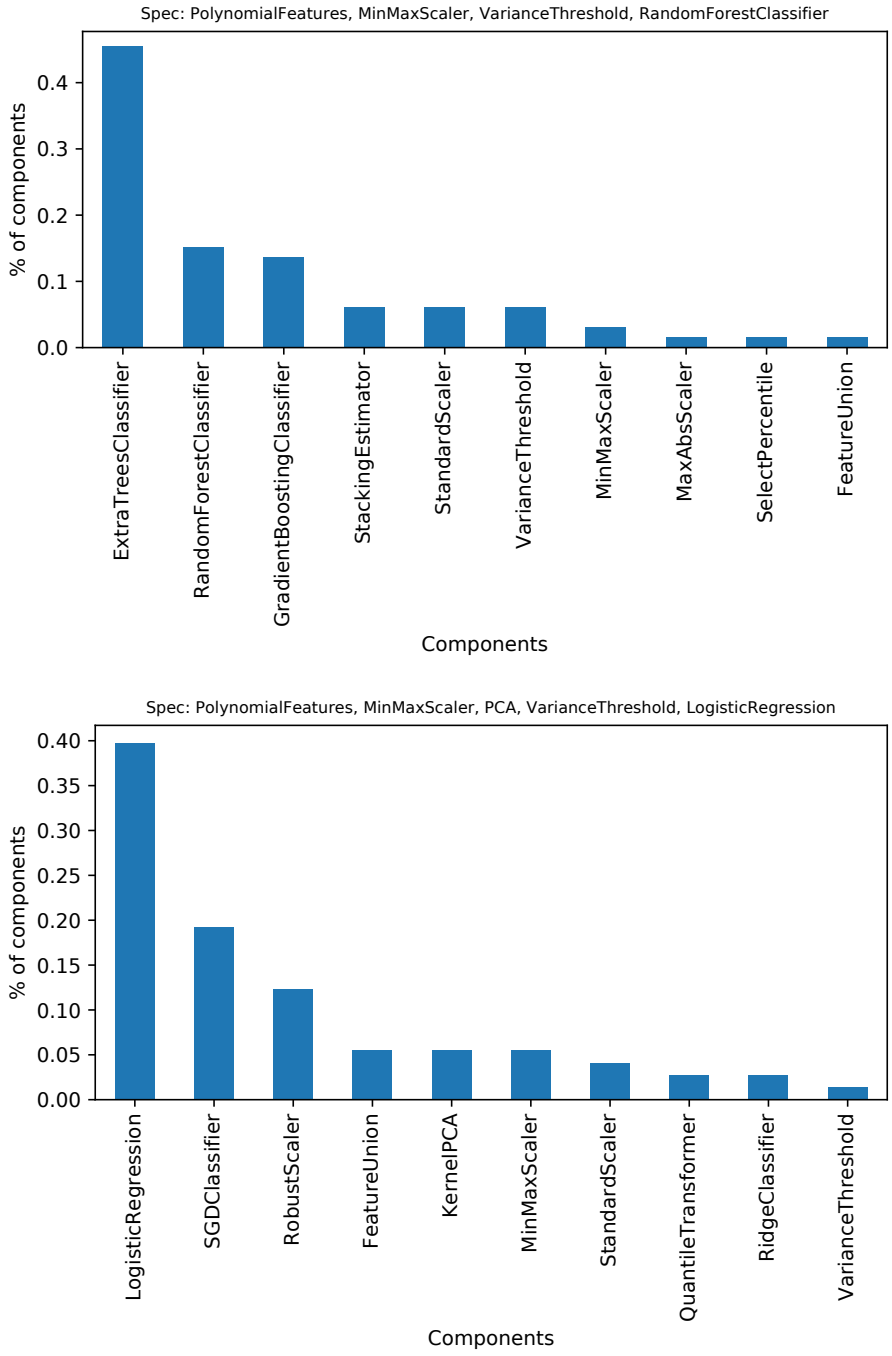


Figure 4-8: Example distributions of scikit-learn components in pipelines produced by genetic programming when using the search-space generated by AMS after strengthening the initial weak specification.

4.5.5 RQ5: Impact of Corpus Size

AMS mines hyperparameters, their corresponding values, and complementary component association rules from a corpus of code examples. To evaluate the impact of varying corpus sizes on AMS, we sampled from 10% to 90% (in 10% increments) of the original 3,300 scripts. We repeated this sampling five times per sampling ratio. For each sampled corpus, we ran AMS's hyperparameter mining and complementary component mining.

Figure 4-9a shows the average fraction of hyperparameters missing for a given component, with respect to the hyperparameters found when using the full corpus. For very small corpora, e.g. 10% (330 scripts), as expected the reduction in hyperparameters mined can be substantial. A moderate sized corpus, e.g. 50% (1650 scripts) covers most of the hyperparameters found in the full corpus.

Figure 4-9b shows the average reduction in possible hyperparameter values with respect to the full corpus. If we mine 5 possible values for a hyperparameter in the full corpus, and we mine 3 possible values in a downsampled corpus, we say that is a reduction of 2 possible values. We see that for a moderate sized corpus (e.g. 50%) the average reduction is approximately one possible value per hyperparameter.

Figure 4-9c shows the decrease in number of complementary components mined, when compared to the full corpus. Small corpora ($< 30\%$ of the original size) display large decreases in the number of association rules found, but moderate sized corpora (e.g. 50%) mine approximately 80% as many rules as the full corpus. Figure 4-9d shows that for moderated sized corpora, the rules mined are relatively similar (approximately 0.8 Jaccard similarity on average) to those mined from the full corpus.

4.6 Threats to Validity

We discuss potential limitations of this research based on design choices. In particular, we focus on threats to generalizability. First, our evaluation uses a particular ML framework (scikit-learn). We believe this threat is mitigated by the fact that scikit-learn is a widely-adopted ML library, used by over 92,000 GitHub repositories as of March 2020. Extending AMS to other popular libraries, such as Tensorflow, may be possible as long as these have

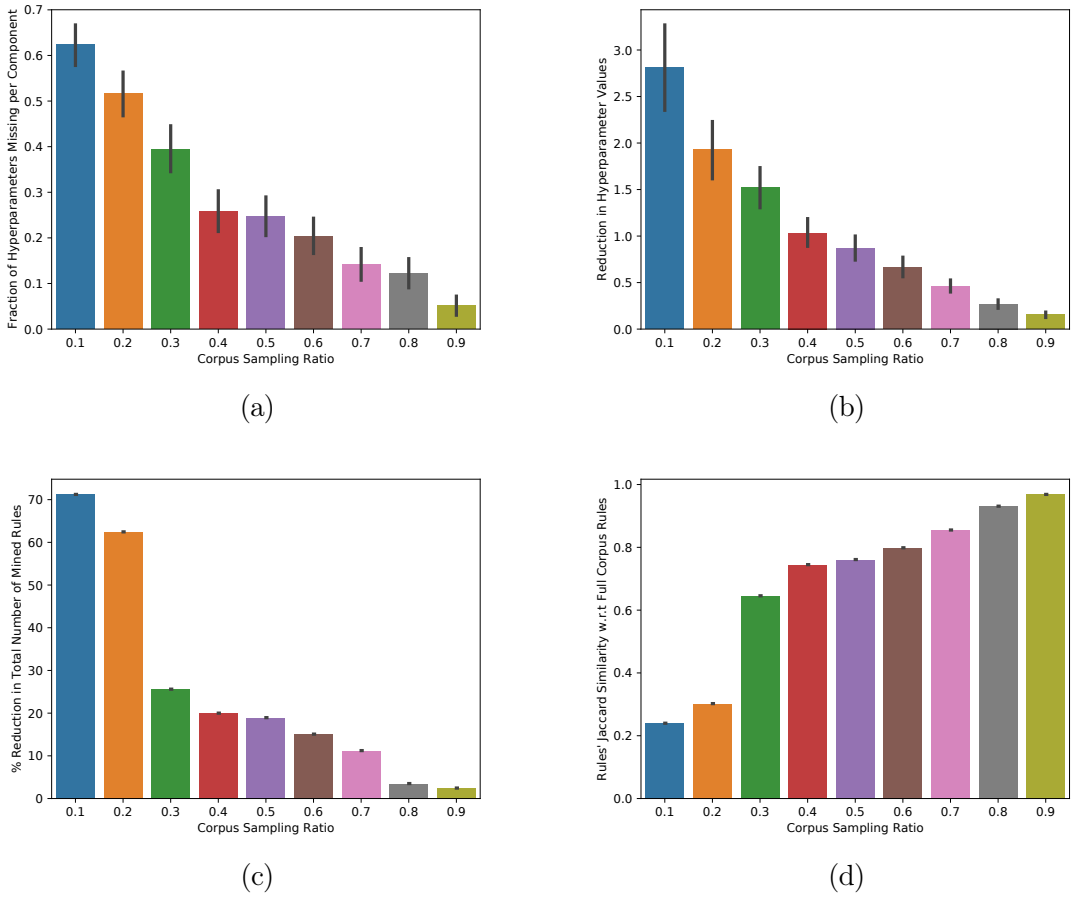


Figure 4-9: Impact of downsampling the full corpus. (a) shows the fraction of missing hyperparameters per component; (b) shows the number of missing possible values per hyperparameter; (c) shows the percent reduction in number of complementary component association rule; and (d) shows the Jaccard similarity of the complementary component rules.

high quality API documentation, with relevant keywords and explanations, and enough online examples for a code corpus[8].

Our code corpus (meta-Kaggle) represents a wide range of scripts written by different users targeting different datasets. Applying AMS to smaller code corpora may impact performance. In our experiments, we found our corpus of approximately 3,300 scripts delivered good performance, and our experiments with varying sizes of code corpus show that a moderate size (approximately 1650 scripts) can deliver reasonably high coverage of hyperparameters and complementary components when compared to our full sized corpus. Further increasing the size of the corpus can help mitigate this risk.

[8]e.g. as of April 28th 2020 72,000 source code projects on GitHub used Tensorflow

We evaluated two search procedures: genetic programming and random search. Other search procedures may potentially find pipelines with different characteristics and performance. However, both random and genetic search are commonly used methods in search-based software engineering and have shown good performance over a wide range of AutoML problems. The choice of evaluation datasets could also influence our results. We used the classification datasets from the original TPOT paper, which have also been used in the evaluation of existing AutoML research [23, 27]. The weak specifications in our evaluation are naturally a sample of possible specifications. However, we aimed to incorporate common operations and components in these specifications to reflect standard usage.

Finally, weak specifications must include at least one task-specific (i.e. regression/classification) component. We believe satisfying this requirement is facilitated by the wide availability of online resources (e.g. tutorials, blogs) describing basic library usage.

4.7 Conclusion

We introduced a new usage model for AutoML, where a user provides a set of API components as a weak specification and this specification can be automatically strengthened. Specifications enable users to exert control and express preferences over the resulting pipeline. We implement our strengthening approach – extending the specification with complementary components using normalized pointwise mutual information on an existing code corpus, functionally related components using a lexical similarity score over the target API's documentation, frequency distributions on constructor calls in the code corpus to extract key hyperparameters and values, and a search procedure – in the AMS system. We evaluated AMS on 9 datasets and 15 weak specifications using two different search procedures. We show that the pipelines produced using AMS's strengthened specifications outperform pipelines produced using the initial weak specifications and variants of the initial specifications annotated with expert-defined hyperparameter spaces.

Chapter 5

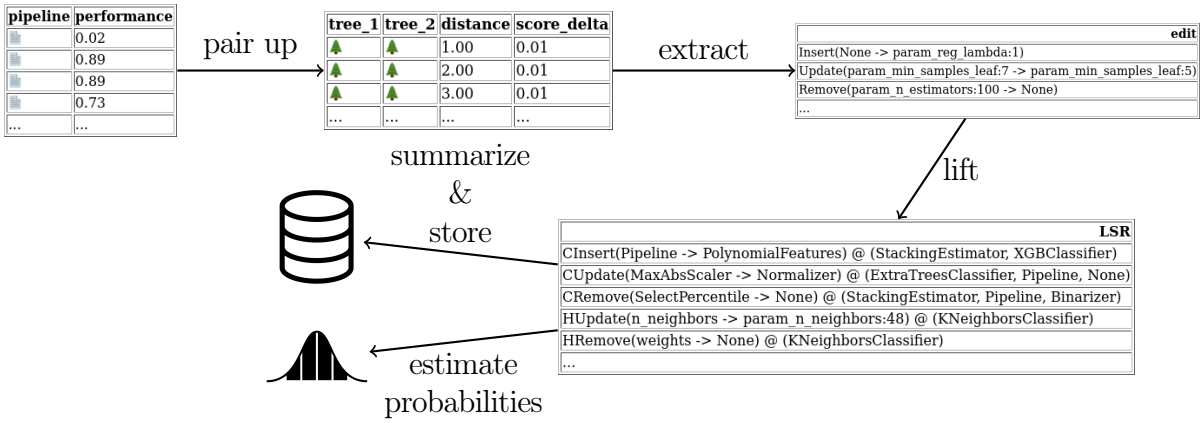
Janus

5.1 Introduction

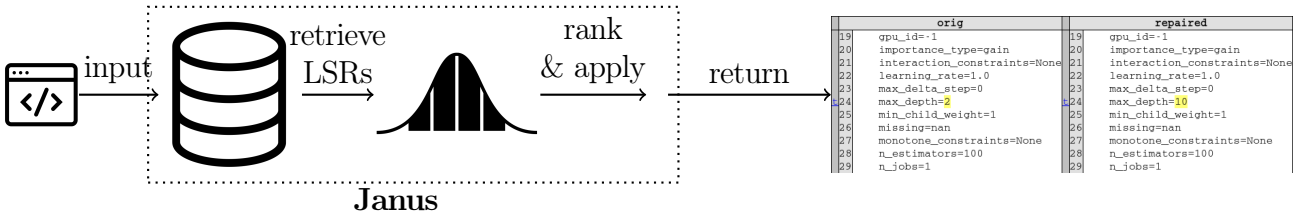
Machine learning pipelines share challenges with other software artifacts, including their maintenance as part of an existing codebase, and their reuse and adaptation as modular software components [4, 58, 103]. However, these challenges are compounded by the difficulties inherent in machine learning, such as varying performance on different datasets, common lack of formal specifications, and the need for background knowledge of the algorithms/operations implemented in the pipeline [77].

A particular challenge arises for developers who are tasked with maintaining an *existing* machine learning pipeline implementation. While improving the pipeline's predictive performance is desirable, so is maintaining a pipeline that does not deviate significantly in design from the original, reducing the footprint of any code changes associated. Small transformations of the pipeline can bring benefits such as the opportunity to better identify the source of performance changes and facilitating faster code review [65].

Directly applying existing automated pipeline search techniques, such as an AutoML tool, in this setting presents two key drawbacks. First, many AutoML tools execute a time-consuming loop of candidate generation and evaluation. Second, most AutoML tools do not take as input an existing pipeline, and if they do (e.g., to warm start a search), they are not constrained to return a pipeline that resembles the original. To address these challenges, we take inspiration from program repair.



(a) In an offline phase, Janus mines transformations from pairs of nearby ML pipelines with a performance differential, extracts concrete edit operations, and lifts these operations to a corpus of abstracted rules, which are then summarized and ranked using observed probabilities.



(b) Given a new pipeline as input, Janus ranks possible transformations and applies them to produce a repair of the original pipeline. In this example, increasing the `XGBoost` classifier hyperparameter `max_depth` value improved the pipeline's F1-macro score from 0.61 to 0.67.

Figure 5-1: An overview of Janus, our system to mine nearby transformations for machine learning pipelines.

A key insight behind this work is that the scenario described closely aligns with traditional automated program repair [76, 74, 75, 64, 7, 48, 67, 78]. In particular, the fact that there may exist a *small* (as of yet unimplemented) modification to the existing pipeline that would improve predictive performance can be viewed as a *bug*. With this perspective, the pipeline modification in turn can be viewed as a *repair*. Automatically identifying and applying this modification, rather than requiring developer intervention is then a natural analogue to *automated program repair*.

We propose Janus, an automated repair system that mines nearby transformations for machine learning pipelines, which when applied can automatically improve their predictive performance. To mine these transformations, Janus first collects a large number of machine learning pipelines and their scores on a set of shared datasets. A key insight in Janus is that if we treat pipelines as trees, we can extract candidate transformations as tree edit operations

from nearby pairs. To this end, Janus defines a d-repair of a pipeline to be a different pipeline with better performance on the same dataset and which is at a tree edit distance of at most d. Janus extracts such d-repairs from the pipeline corpus to use as inputs to its transformation mining procedure. When extracting these pairs, Janus efficiently prunes candidates down by using an approximation of the tree edit distance as a filter [59]. Edit operations extracted from the final set of tree pairs are then lifted to an abstraction we term *local structural rules*, a typed version of edit operations with ML pipeline specific semantics. Janus summarizes the transformation rules observed into a rule corpus, over which it can compute the joint probability of a given rule and the tree node at which it is applied. Given a new pipeline, Janus returns a repaired pipeline produced by greedily applying the most likely transformation that results in a new pipeline within d edits of the original pipeline. This approach is conceptually similar to existing learned program repair techniques [74, 7, 67].

To evaluate Janus, we collect a corpus of pipelines generated using an off-the-shelf genetic programming AutoML tool, TPOT [85]. Using the same tool, we generate 100 different test pipelines for 9 different datasets. We evaluate Janus's ability to produce d-repairs for these 900 input pipelines, and compare to three baselines.

Our results show that Janus can improve $16\% - 42\%$ of the pipelines across our test datasets, more than baseline approaches in 7 of our 9 datasets. We also evaluate our system design. We show that when extracting tree pairs, Janus' approximate distance metric effectively reduces tree pairing runtime while producing results comparable to an exact approach. We also show that Janus can effectively summarize repair rules, reducing an original set of transformations by 92.7%. Finally, we carry out a sensitivity experiment, where we replace the corpus of pipelines with pipelines produced using random search. Janus now mines transformations and evaluates repairs from this random search-based corpus. In this setting, we show that Janus can still improve $24\% - 43\%$ of test pipelines, but now only outperforms baselines in 2 of our 9 datasets, highlighting the importance of the pipeline corpus.

5.1.1 Janus Overview

Figure 5-1 presents an overview of Janus's two phases. In an offline phase (Figure 5-1a), the system mines repair rules, which are then applied in an online phase (Figure 5-1b) to

produce repairs.

Building a Pipeline Corpus. The process of mining repair rules starts by collecting a large number of machine learning pipelines and identifying pipelines that are "nearby" but have a performance differential. To formalize this insight, we treat pipelines as trees and employ tree edit distance to quantify the degree to which they are related. Janus uses a two step procedure, which first identifies candidate tree pairs based on an approximate distance metric [59] and then refines this using the exact tree edit distance measure to efficiently collect pairs of pipelines.

From Edits to Local Structural Rules. As the next step in the offline phase, Janus extracts the sequence of edit operations (e.g. remove, update, insert) that transform the lower scoring pipeline in a pair into the higher scoring pipeline. However, tree edit operations have key limitations: they are defined only in the context of the tree pair from which they are extracted, and they are generic in that they do not account for machine learning pipeline semantics. Janus addresses this challenge by lifting simple edit operations to *local structural rules* (LSR), which are typed edit operations with both pre- and post-conditions which validate whether a rule can be applied to a new node. At this point in the procedure, Janus takes the collection of observed LSRs, abstracts them using a key-based approach (Section 5.4), and summarizes them by keeping a single rule per shared key. While summarizing, Janus computes two probability distributions: the conditional probability of applying a transformation with a particular rule key given a tree node with a particular label, and the marginal probability applying a transformation at a node with a particular label. Janus stores the summarized rule corpus and the two distributions for use in the online phase.

Rule-based Repairs. In its online phase, Janus takes as input a machine learning pipeline. It retrieves the set of possible LSRs for each node and ranks the (LSR, node) candidates based on their joint probability, which we compute as the product of the conditional and marginal probabilities collected during the offline phase. Janus implements a lazy tree generator to greedily enumerate transformed trees. Janus returns as a repair the *first* transformed tree that satisfies the user-specified distance bound and does not generate any runtime exceptions. In particular, note that Janus *does not* repeatedly generate, fit,

and evaluate candidate pipelines.

The remainder of the chapter is structured as follows. The background and core of Janus is described from Section 5.2 to Section 5.4. We describe our evaluation methodology in Section 5.5, our results in Section 5.6, and threats to validity in Section 5.7. Section 5.8 concludes.

5.2 Building a Pipeline Corpus

5.2.1 Pipelines as Trees

Janus represents machine learning pipelines as trees. Pipeline trees are defined as a set of typed nodes, $\mathcal{V}$, and a directed edge function $E : \mathcal{V} \times \mathcal{V} \to \mathbb{B}$ that returns true if there is a directed edge from the first to the second node. For brevity, we use the type $\mathcal{T}$ to denote tree representations and the type $\mathcal{F}$ to denote the space of pipelines.

Nodes in a tree can be of two types: component nodes ($\mathcal{C} \subset \mathcal{V}$) or hyperparameter nodes ($\mathcal{H} \subset \mathcal{V}$). A component node represents an API component from the third-party library used to implement the machine learning pipeline. For example, `LogisticRegression` can be represented with a component node. Component nodes can further be split into two subtypes: combinator components, which represent composition (e.g. applying components in series or joining the results of one or more subtrees), or non-combinator components. A hyperparameter node represents the tuple (hyperparameter, value) defined for a particular API component. For example, a regularization weight set to value 1.0 can be represented with a hyperparameter node.

Janus defines a few standard functions over nodes in the tree. The label function, $\textsc{label} : \mathcal{V} \to \text{string}$, produces a label for a node. Component node labels are defined as the fully qualified path for the API component they represent. Hyperparameter node labels are defined as the string concatenation of their hyperparameter name and their value. The PARENT, LEFT and RIGHT sibling functions, of type $\mathcal{V} \to (\mathcal{V} \cup \emptyset)$, and children function $\textsc{children} : \mathcal{V} \to \mathcal{P}(\mathcal{V})$, produce the expected set of nodes based on tree traversals. Hyperparameter nodes also define a function $\textsc{Value} : \mathcal{H} \to (\mathbb{R} \cup \mathbb{B} \cup \text{string})$ which returns

the underlying value of that hyperparameter in the pipeline definition.

Janus defines two (bijective) functions to transform pipelines into trees, and vice versa. ToTree : $\mathcal{F} \rightarrow \mathcal{T}$ maps a pipeline to its tree representation, with inverse FromTree:$\mathcal{T} \rightarrow \mathcal{F}$.

Janus defines the distance between two pipelines, Dist:$\mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R}$, as the tree edit distance [28] between their respective tree representations. When comparing nodes in the tree edit distance computation, Janus uses binary distance over the node labels.

The goal of Janus is to improve the performance of a pipeline by applying a transformation that yields a "nearby" pipeline. We call this a d-repair.

Definition 3 *d-repair. Let f be a pipeline, X and y be a training dataset, X' and y' be a test dataset, e be an evaluation function where a higher value is better, and let $f(X,y)(X')$ be shorthand for training pipeline p on X and y and predicting outputs for X'. We say f' is a d-repair if* $\text{Dist}(f,f') \leq d \wedge e(f(X,y)(X'),y') < e(f'(X,y)(X'),y')$.

We refer to f as the pre-pipeline and ToTree(f) as the pre-tree. Similarly, we refer to f' as the post-pipeline and ToTree(f') as the post-tree. We refer to the pair (f,f') as a d-repair.

5.2.2 Corpus of Tree Pairs

Janus is designed to mine repair rules that can be applied to a pipeline to produce a d-repair. To extract these rules, Janus first requires a large corpus of machine learning pipelines that have been scored on a shared dataset. To collect this initial set of machine learning pipelines, Janus runs an off-the-shelf automated pipeline search procedure, which generates and evaluates many candidate pipelines, all of which Janus collects.[1] Using an AutoML system to produce a pipeline corpus also has the added benefit of covering regions of varying performance and pipeline distances, increasing the likelihood of productive d-repairs for rule mining. In principle, Janus could collect a corpus through alternative approaches such

[1]Most AutoML tools return a single or small set of optimized pipelines, but Janus instead collects *all* the pipelines the tool encountered during its search.

as mining experiment tracking frameworks like ModelDB [117] or collaborative machine learning websites like OpenML [116].

Once a large collection of pipelines has been collected, Janus samples d-repairs to use for its rule extraction procedure.

Algorithm 7 illustrates our approach to constructing a corpus of observed d-repairs. Janus first samples N_{query} pipelines uniformly at random from our pipeline corpus. We refer to these pipelines as *query pipelines*. The goal is to pair each query pipeline with a set of at most k d-repairs.

Computing the edit distance for all pairs of pipelines in the corpus is expensive, with the distance operation having time complexity of $\mathcal{O}(n^2 m(1+log\frac{m}{n}))$ given a tree with n nodes and another with m nodes [28]. To address this challenge, Janus exploits the fact that a key property of d-repairs is better predictive performance, and first filters candidates down to those that have higher score. Janus then further refines this set of candidates to the top N_{maybe} pipelines that are *possibly* a d-repair based on an approximate distance metric with linear time complexity.

To compute an approximate distance metric (Algorithm 8), we first take the tree representation of a pipeline and flatten it into a string representation, akin to an S-expression. This string representation is tokenized by splitting the string on any non-alphanumeric characters and making parentheses tokens as well (as their count correlates with tree structure). Finally, we compute a vector of token counts, where an entry is set to the number of times the token appeared in the string. The approximate distance between two trees is then defined as the Euclidean distance between their count vectors. Janus takes the N_{post} candidates with the smallest approximate distance to the query pipeline under consideration. This use of a token-based vector representation for distance approximation is similar to the use of characteristic vectors in DECKARD [59], but our "characteristic" patterns are restricted to individual token occurrence counts.

Finally, Janus computes the exact tree-edit distance on this smaller set of candidate pipelines, and keeps pipelines that are at most a distance d from the query pipeline. We pair the query pipeline with each of these pipelines to produce an observed d-repair. Janus repeats this process for each of the N_{query} pipelines initially sampled.

Algorithm 7 Sampling d-repairs for a single query pipeline to build up a corpus for rule mining

INPUT: A corpus C of tuples, where each tuple is a machine learning pipeline and its corresponding test score; a function ApproxDist to compute the approximate distance metric between two pipelines; a function Dist to compute the exact tree edit distance between the tree representations of two pipelines; a query pipeline in tree form t_q and its performance score s_q; an integer N_{maybe} for the number of *potential* d-repairs to sample; an integer d for the maximum tree edit distance for a d-repair; and an integer k for the maximum number of d-repairs to produce per query pipeline.

OUTPUT: At most k d-repairs

procedure SampleTreePairs

▷ *d-repairs must have better score*

$candidates \leftarrow \{t \mid (t,s) \in C \setminus (t_q,s_q) \wedge s > s_q\}$

▷ *Prune down using approximate distance as sorting function*

$candidates \leftarrow$ SortBy($candidates$, λt_i: ApproxDist(t_i,t_q))

$candidates \leftarrow$ Take($candidates$, N_{maybe})

▷ *Enforce distance threshold, to guarantee these are d-repairs*

$repairs \leftarrow \{t \mid \text{Dist}(t,t_q) \leq d\}$

$repairs \leftarrow$ Take($repairs$, k)

return $\{(t_q,t) \mid t \in repairs\}$

end procedure

Algorithm 8 Approximate Distance Metric

INPUT: Tree representations t_1 and t_2 of two pipelines.

OUTPUT: An approximate distance between two trees

procedure ApproxDist

▷ *Get string representation of trees*

$s_1 \leftarrow$ ToString(t_1)

$s_2 \leftarrow$ ToString(t_2)

▷ *Vectorize string representations as count of tokens*

▷ *Includes structural tokens like parentheses*

$v_1 \leftarrow$ Vectorize(s_1)

$v_2 \leftarrow$ Vectorize(s_2)

▷ *Return euclidean distance*

return EuclideanDist(v_1,v_2)

end procedure

5.3 Local Structural Rules

In Janus, we propose that we can repair a pipeline by learning from the operations required to transform pre-pipelines to post-pipelines in d-repairs observed in our pipeline corpus. These basic transformation operations are defined to be the sequence of update, insert, and delete operations computed for purposes of the tree edit distance [28]. An update

operation updates a node in the tree with a new label, an insert operation inserts a new node in a tree, and a delete operation removes a node in a tree.

While edit operations are useful information, they do not represent transformation rules. In particular, a given edit operation is only defined over the two input trees used to compute the overall sequence of edit operations. And importantly, these edit operations are generic, defined for any tree representation, but lack the semantics specific to tree representations of machine learning pipelines.

To bridge this gap, Janus introduces an abstraction: a *local structural rule* (LSR). Let $\mathcal{L}$ be the set of possible rule types HUpdate (update a hyperparameter node), HRemove (remove a hyperparameter node – equivalent to setting the original default value provided by the API), CUpdate (update a component node), CRemove (remove a component node), and CInsert (insert a component node).

Definition 4 *Local structural rule.*

We define a local structural rule as a triple in $\mathcal{L} \times \mathcal{T} \times \mathcal{T}$, where the first element is the rule type, the second is the pre-tree, rooted at the target location of the transformation, and the third is the post-tree, rooted at the output location of the transformation.

An LSR has important differences compared to an edit operation. LSRs are typed, meaning there is a distinction between update/insert/remove operations based on whether the node it applies to is of component type or hyperparameter type. LSRs contain pre- and post-conditions, that relate the transformation to the tree pair from which it was mined. Pre-condition predicates rely on the *pre-tree*, and post-condition predicates rely on the *post-tree*. These pre/post-conditions are particularly useful as they allow LSRs to implement a CanApply: $\text{LSR} \times \mathcal{V} \rightarrow \mathbb{B}$ predicate which validates whether a transformation can be applied to a given tree node. Finally, LSRs are *local* in nature, meaning the conditions checked can access the candidate application node itself and other nodes with which it shares a direct edge in the tree.

Figure 5-2 presents Janus' LSRs in terms of inference rules. In general, an LSR checks that the node is of the appropriate type (i.e. $\mathcal{C}$ or $\mathcal{H}$), is not a no-op change (where the node already has the value that would result from the rule application), and has appropriate

neighbors. HUpdate and HRemove also include a check on the current value of the hyperparameter (CheckValue), which requires exact equality between the current node's value and the LSR's pre-tree value when the current node's value is of type string. The intuition here is that string values tend to indicate categorical options in ML pipelines (e.g. a type of penalty or kernel) and as such an LSR should only apply if it is the same value, while continuous values need not be exact for an LSR to productively modify it. For the purpose of inserting a new component node, Janus samples a position in the target node's children. To apply an LSR, Janus leverages two tree structure helpers: Replace and Insert. Both of these functions take as arguments a tree, an existing node or a position, and a new node, and return a new version of the tree where we have placed the new node at the indicated position. Both of these functions incorporate ML-specific pipeline semantic checks and transformations, such as: ensuring the last operator in a pipeline is a classifier, ensuring that any classifiers inserted/updated into earlier positions in the pipeline are wrapped to stack their predictions as a new feature, and pruning out any empty subtrees.

5.4 Rule-based Repairs

To effectively use the LSRs lifted from the collection of edit operations in our corpus, we must abstract and summarize them. Janus carries out this summarization process using partial information over rules and a greedy heuristic. Next, Janus uses a joint probability distribution computed from the observed edits in our d-repair corpus to rank pairs of the form (rule, target node) in a new input tree. This ranking is used by a lazy tree generator, which produces candidate transformed trees consumed by the core repair algorithm. We now provide the details of this process.

5.4.1 Abstracting Local Structural Rules

At this point in the operation of Janus we have extracted a collection of LSRs, derived from the sequence of edit rules applied to transform the corpus of tree pairs from pre-trees to post-trees. These LSRs are effectively a corpus of observed rule applications. To perform future repairs, Janus has to organize and summarize this corpus. For this we introduce

$$\frac{t\in\mathcal{T} \quad n\in\mathcal{H} \quad (\textsc{HUpdate},t_{\text{pre}},t_{\text{post}}) \quad \textsc{Label}(\textsc{Parent}(n))=\textsc{Label}(\textsc{Parent}(t_{\text{pre}})) \quad \textsc{CheckValue}(\textsc{Value}(n),\textsc{Value}(t_{\text{pre}})) \quad n\neq t_{\text{post}}}{\textsc{Replace}(t,n,t_{\text{post}})}$$

$$\frac{t\in\mathcal{T} \quad n\in\mathcal{H} \quad (\textsc{HRemove},t_{\text{pre}},\emptyset) \quad \textsc{Label}(\textsc{Parent}(n))=\textsc{Label}(\textsc{Parent}(t_{\text{pre}})) \quad \textsc{CheckValue}(\textsc{Value}(n),\textsc{Value}(t_{\text{pre}}))}{\textsc{Replace}(t,n,\emptyset)}$$

$$\frac{\begin{array}{c}t\in\mathcal{T} \quad n\in\mathcal{C} \quad (\textsc{CUpdate},t_{\text{pre}},t_{\text{post}}) \\ (\textsc{Label}(\textsc{Parent}(n))=\textsc{Label}(\textsc{Parent}(t_{\text{pre}}))\vee \quad (\textsc{Label}(\textsc{Left}(n))=\textsc{Label}(\textsc{Left}(t_{\text{pre}}))\vee \\ (\textsc{Label}(\textsc{Right}(n))=\textsc{Label}(\textsc{Right}(t_{\text{pre}}))) \quad n\neq t_{\text{post}}\end{array}}{\textsc{Replace}(t,n,t_{\text{post}})}$$

$$\frac{t\in\mathcal{T} \quad n\in\mathcal{C} \quad (\textsc{CRemove},t_{\text{pre}},\emptyset) \quad (\textsc{Label}(\textsc{Parent}(n))=\textsc{Label}(\textsc{Parent}(t_{\text{pre}}))\vee \quad (\textsc{Label}(\textsc{Left}(n))=\textsc{Label}(\textsc{Left}(t_{\text{pre}}))\vee \quad (\textsc{Label}(\textsc{Right}(n))=\textsc{Label}(\textsc{Right}(t_{\text{pre}})))}{\textsc{Replace}(t,n,\emptyset)}$$

$$\frac{\begin{array}{c}t\in\mathcal{T} \quad n\in\mathcal{C} \quad (\textsc{CInsert},t_{\text{pre}},t_{\text{post}}) \quad \textsc{IsCombinator}(n) \\ \textsc{Children}(n)\neq\emptyset \quad \{\textsc{Label}(c)|c\in\textsc{Children}(n)\}\cap\{\textsc{Label}(c)|c\in\textsc{Children}(t_{\text{pre}})\}\neq\emptyset\end{array}}{\textsc{Insert}(t,\textsc{SampleChildPosition}(n),t_{\text{pre}})}$$

Figure 5-2: Local structural rules (LSRs) are typed edit rules used by Janus to perform tree transformations. The rules provide pre-conditions that verify whether a rule can be effectively applied to a node. These pre-conditions, as well as the tree helper functions Replace and Insert, leverage semantics specific to ML pipelines.

the concept of an LSR key.

Definition 5 *LSR key.*

An LSR key is a function $\textsc{Key}: \text{LSR} \rightarrow \mathcal{L} \times \textit{string} \times \mathcal{P}(\textit{string}) \times \textit{string}$*, which given an LSR returns a 4-tuple consisting of the type of the corresponding LSR, the label of the pre-tree root node, an (unordered) set of labels over a subset of neighbors (*context*), and the label of the post-tree root node.*

For hyperparameter LSRs (HUpdate, HRemove), the context is the parent label. For CUpdate and CRemove, the context is the labels for the parent, left, and right sibling nodes. For CInsert, the context is the set of labels of its children.

We assign a score change to every LSR mined. In particular, we assign each LSR the score change associated with the d-repair that produced it. Note that while the tree pair

may induce multiple edit operations (and thus multiple LSRs), every LSR derived from the pair is assigned the same score difference.

Given a collection of LSRs, Janus greedily summarizes the collection by keeping the LSR with the highest score for a given LSR key. This heuristic use of scores is meant to identify rule instances that are likely to induce a performance change. We refer to this summarized collection of LSRs as the rule corpus.

While building the rule corpus, Janus computes two key statistics over the original collection of LSRs: the conditional probability of observing an LSR key given a pre-tree node label (denoted as P(rule-key|node)) and the marginal probability of observing a given pre-tree node label over all rules (denoted as P(node)). Both of these probabilities can be computed by simply counting and normalizing appropriately.

5.4.2 Ranking and Applying Rules

Janus collects a map from pre-tree label to the set of LSRs with that corresponding pre-tree label[2] in the summarized corpus. Janus uses this node-to-rule-set map to retrieve a set of potentially relevant rules, when given a node. To produce a sorted list of candidate tree transforms, Janus traverses an input tree, accumulates a collection of possible (LSR, node) pairs, and then sorts these based on their joint probability. The sorted list represents Janus' ranking of LSRs and target application location, each of which constitutes a possible repair. This procedure[3] is summarized in Algorithm 9.

Janus, given an input tree, produces a (lazy) tree generator which a downstream repair step can query for candidate transformed trees. This tree generator yields a new candidate tree by applying each of the (rule, node) pairs in their ranked order, and checking if the new candidate tree can be used to successfully build a pipeline object. This procedure is summarized in Algorithm 10.

To repair a pipeline, Janus takes the original input pipeline and uses it to initialize the tree generator. The repair loop then requests a tree, checks whether it is within the

[2]For hyperparameter-related LSRs, the pre-tree label does not include the hyperparameter value, just its name.

[3]We explicitly factor out the tree traversal into a separate step for clarity, but our implementation fuses these steps.

Algorithm 9 Generating ranked list of LSR and tree location for tree transformation.

INPUT: A tree t; a node-to-rule-set map R; a marginal probability function MARGINALPROB that computes P(node); a conditional probability function CONDPROB that computes P(rule-key|node); a function KEY that retrieves the LSR Key for a rule; Janus's predicate function CANAPPLY which validates an LSR's pre-conditions over a concrete tree node.

OUTPUT: A list of (LSR, node) entries ranked in descending order of joint probability.

```
procedure RankTreeTransformations
    ▷ Nodes in tree are possible locations for transform
    N ← CollectNodes(t)
    ▷ Retrieve possible LSRs based on node
    candidates ← {(r,n) | n ∈ N,  r ∈ R(n)}
    ▷ Remove LSRs that can't be applied based on pre-conditions
    candidates ← {(r,n)| (r,n) ∈ candidates ∧ CanApply(r,n)}
    ▷ Sort with joint probability function
    ranked ← SortBy(candidates, λ(r,n): CondProb(Key(r),n)*MarginalProb(n))
    return ranked
end procedure
```

Algorithm 10 Janus lazy tree enumerator.

INPUT: An input tree t; Janus's CANCOMPILE which tries to lower a tree to its pipeline representation (using FROMTREE) and returns true if it succeeds without any pipeline building exceptions; Janus's RANKTREETRANSFORMATIONS function to produce a ranked list of transformations and their location; and Janus's APPLY function which takes a tree, a node location, and applies a rule to it.

OUTPUT: A lazy generator for transformed trees.

```
procedure TreeGenerator
    ▷ Queue of derived trees, starts with input
    transforms ← RankTreeTransformations(t)
    for (r,n) ∈ transforms do
        h' ← Apply(h,n,r)
        if CanCompile(h') then
            yield h'
        end if
    end for
    ▷ null tree as sentinel
    return ∅
end procedure
```

pre-specified distance bound d, tests whether the new tree produces a runtime exception on a small sample of the user's dataset, and if no exception is raised it returns the associated pipeline as a repair. To avoid situations where the tree generator may fail to produce a d-repair but continues to yield candidate transformations, Janus takes a time budget (set to 60 seconds by default). If no repair validates during this time, Janus returns a null

pipeline. Algorithm 11 summarizes this repair procedure.

Algorithm 11 Janus high-level d-repair procedure.

INPUT: A pipeline f; Janus's ToTree and FromTree functions mapping pipelines to trees and vice-versa; Janus's TreeGenerator function yielding (on request) transformed trees; a function Dist that computes exact tree edit distance; a function Fit which attempts to fit the pipeline to a sample dataset; an integer bound d on the maximum tree edit distance for a candidate repair; a sample of the user's dataset (X,y); a function TimeSoFar that indicates how much time has elapsed, and a time limit b (default to 60 seconds).

OUTPUT: Janus's repair for the input pipeline

```
procedure Repair
    t ← ToTree(f)
    ▷ Instantiate lazy tree generator
    gen ← TreeGenerator(t)
    ▷ Limit repair time for responsiveness
    while TimeSoFar() < b do
        ▷ Next tree in the generator's queue
        t' ← gen.next()
        ▷ null pipeline if no more transforms possible
        if t' == ∅ then
            return ∅
        end if
        ▷ Repair too far away
        if Dist(t,t') > d then
            continue
        end if
        ▷ Lower to pipeline
        f' ← FromTree(t')
        try
            ▷ Check possible exceptions on sample data
            Fit(f',X,y)
            return f'
        catch Exception
            continue
        end try
    end while
    ▷ null pipeline if no repair produced
    return ∅
end procedure
```

5.4.3 From Scripts to Pipelines

In practice, machine learning pipelines are often written as part of larger ad-hoc experimental scripts or computational notebooks [111]. These artifacts will typically perform

additional steps, beyond just building a pipeline. For example, it is common (and good practice) for users to visualize the dataset they are working on, explore deriving new features, and validate different model choices. To facilitate use of Janus in such a setting, we have implemented a front-end to Janus, which supports extracting the subset of code involved in the definition of the machine learning pipeline. This front-end allows a user to extract a pipeline, apply Janus, and obtain a repaired pipeline.

To build this front-end, we rely on program instrumentation and dynamic analysis. Specifically, we target scripts/notebooks written in Python, leveraging Python's built-in debugging hooks (`sys.settrace`). Our front-end first converts Jupyter notebooks to scripts, if necessary. We then extract a source-line-level dependency graph based on executing the program and tracking the memory address of definitions and uses.[4] Janus's front-end identifies nodes in the graph involving our target ML library (scikit-learn). Within these nodes, the front-end uses method name matching to identify calls to the prediction method of any classifier. These nodes become seed nodes for a backwards slice through the graph. For each such slice, we then re-execute each node (in topological order based on the directed edges of the dependency graph) and record the concrete scikit-learn object instantiated, each such object becomes a step in our lifted pipeline. At the end of this procedure, the front-end returns one (or more) pipelines constructed based on the script contents. These pipelines are then given to Janus's repair module, as detailed previously.

5.5 Experimental setup

We evaluate Janus on several dimensions, focusing on its ability to repair pipelines. In particular, we compare the effectiveness of different approaches in producing d-repairs (Definition 3). For our evaluation, we set $d = 10$, a distance bound that is large enough to allow full pipeline component changes, but small enough to reflect the original input pipeline. We now describe our experimental setup.

[4] This is an approximate procedure, and relies on CPython's `id` behavior.

5.5.1 Pipeline corpus

For our evaluation, we use the nine datasets in the TPOT evaluation corpus [82]. For each dataset, we produce a pipeline corpus by running TPOT [85], a genetic programming AutoML tool, and collecting all the candidate pipelines generated during the tool's search.

For each dataset, we use 50% of the data as a development set. The other 50% of the data is held-out to be used as a test set for evaluating repairs produced.

For each development set, we run TPOT for two hours using its default configuration to produce a pipeline corpus. Search is carried out on 80% of the development set. For each pipeline generated we compute its score on the remaining 20% of the development set using macro-averaged F1. We obtain a total of 19,169 pipelines paired with their scores on the validation set as a pipeline corpus.

5.5.2 Extracting tree pairs

From this pipeline corpus, we extract d-repairs (Section 5.2.2) for Janus to mine rules. For each dataset, we sample scored pipelines and convert them to corresponding tree representations, collecting 200 pre-trees. For each pre-tree, we sample pipelines that produced a higher score, collecting 50 post-trees. We keep (at most) the $k = 10$ closest post-trees for each pre-tree, resulting in a total of 16,778 tree pairs, and then extract rules from pairs that satisfy our distance bound.[5]

5.5.3 Extracting rules

We extract rules from our d-repairs and summarize these rules to compute a rule corpus following the approach described in Section 5.3 and Section 5.4. This results in 40,232 raw LSRs lifted from edit operations, which Janus summarizes to produce a rule corpus of 2,939 rules.

[5] A pipeline may have fewer than 10 other d-repairs in our corpus, thus the total tree pairs is less than 18,000.

5.5.4 Baselines

We compare Janus to three alternative strategies for producing d-repairs.

- *Random-Mutation*: Generates repair candidates by randomly sampling a tree node and a corresponding tree edit operation. Tree edit operations are parameterized based on the search space defined in TPOT.

- *Janus-Random*: Randomizes the application of Janus-mined rules.

- *Meta-Learning*: A strategy inspired by task-independent meta-learning. When a repair candidate is requested, this approach queries *Random-Mutation* for $k = 5$ random mutations, scores them using a predictive score model, and puts them into a priority queue (with their predicted score as a sorting key) from which it returns the highest scored candidate available. The predictive score model takes a pipeline, encodes it using the vector-based representation introduced in Algorithm 8, and uses a random forest regression model to predict the corresponding score. We use the random forest regression implementation available through scikit-learn [89] (version 0.22.2) with default hyperparameters.

5.5.5 Producing candidate repairs

Each approach is given access to 5% of the development set to validate that a candidate repair does not produce a runtime error. Every system returns the *first* pipeline to produce no runtime errors and satisfy the distance constraint of $d = 10$. If no such repair is found within 60 seconds, the system returns a null pipeline (meaning no repair candidate was found).

5.5.6 Evaluating candidate repairs

For each dataset, we sample 100 pipelines from the pipeline corpus (Section 5.5.1) and produce a candidate repair with each approach. We evaluate each candidate repair on the held-out test set by computing its macro-averaged F1-score (ranging from 0 to 1.0) using 5-fold cross-validation. If no pipeline is produced, we record a `nan` score.

We say a repair had an effect on pipeline performance if the absolute difference of the repair's macro-averaged F1-score to the original score is at least 0.01. We say a repair improved a pipeline if it had an effect and the score change was positive. We say a repair hurt a pipeline if it had an effect and the score change was negative.

To avoid leakage when producing candidate repairs, we blind all approaches to any pipelines associated with the dataset under consideration. This means that *Meta-Learning* trains its score model only on pipelines associated with other datasets, and Janus and *Janus-Random* only use rules derived from pipelines associated with other datasets.

5.6 Evaluation

We present the evaluation of Janus design choices, repair performance, distance of repairs, and importance of the underlying pipeline corpus.

5.6.1 Janus Design

We evaluate the impact of the approximate distance metric (Algorithm 8) on the distance distribution for pairs of trees collected by Janus for rule mining. In particular, we compare the use of an approximate distance metric in tree sampling to a uniform random sampling approach and an exact approach. For the exact approach, we compute the exact tree edit distance for all pairs of pipelines that have a higher score than our query pipeline and take the top 10 closest pipelines. For the uniform random sampling approach, we take all pipelines that have a higher score than our query pipeline and then randomly sample 50 of these, compute the exact tree edit distance for this subset, and then take the top 10 closest pipelines. Figure 5-3 shows the empirical cumulative distribution function (ECDF) for the distance between tree pairs produced using all three methods. Our results show that the approximate distance approach strictly improves on random sampling, and very closely matches the results obtained with the exact approach. The success of this approximation in our context aligns with the results derived by Jiang et al [59] and Yang et al [123].

In our experiments the approximate distance approach had an average runtime of 17.5 ($\sigma = 6.95$) minutes per dataset, compared to an average of 360.76 ($\sigma = 253.33$) minutes for

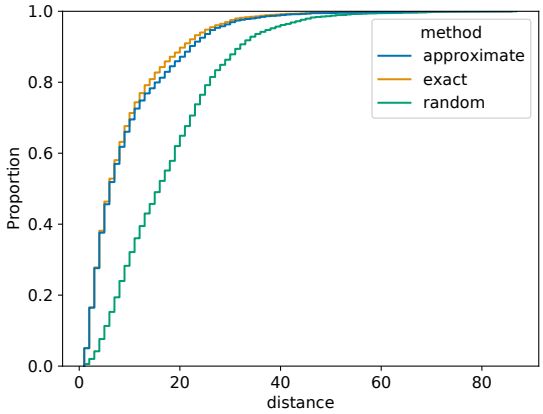


Figure 5-3: The approximate distance metric used by Janus to efficiently identify d-repair candidates for rule mining. This approximation closely tracks an exact distance method, while being substantially faster. This result aligns with that derived by Jiang et al [59] and Yang et al [123],

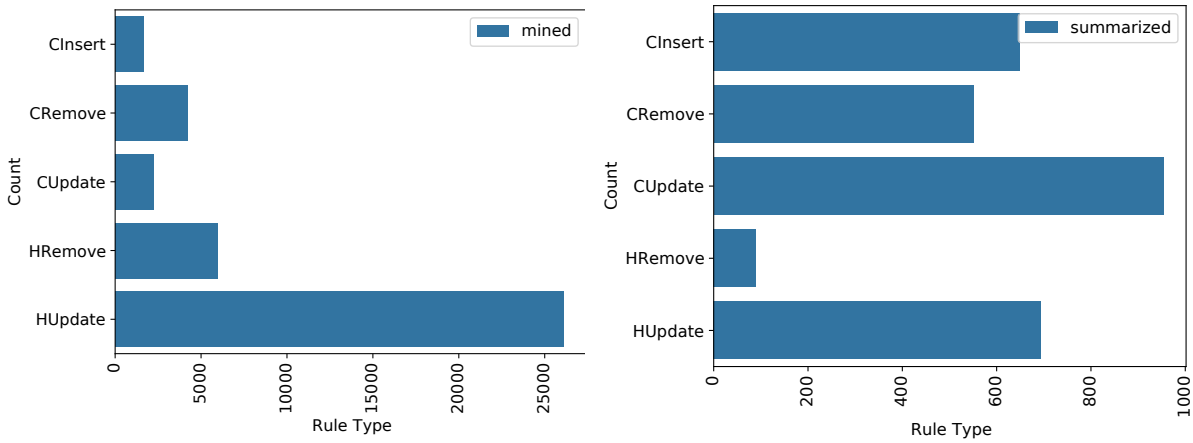
the exact method and an average of 18.72 ($\sigma = 7.06$) minutes for the random approach.[6]

To produce a rule corpus, Janus first lifts edit operations to local structural rules (LSRs). In this process, Janus mines a total of 40,232 LSRs. To effectively generate tree transforms, Janus summarizes this set of LSRs to a final corpus to 2,939 rules, relying on the LSR keys (Definition 5) and a greedy score heuristic. This summarization reduces the initial set of LSRs by 92.7% and normalizes the initially skewed distribution of LSR types from a large number of HUPDATE rules to a more balanced mix. Figure 5-4 illustrates the distribution of LSR types after summarization.

5.6.2 Performance

Next we evaluate Janus' ability to improve pipelines through repairs. Table 5.1a shows the fraction of pipelines where the candidate repair improved over the original, along with bootstrapped 95% confidence intervals. Janus improves the performance of 16%–42% of our test pipelines, outperforming baseline approaches in 7 of our 9 datasets. *Meta-Learning* produces more successful repairs in two datasets. Table 5.1b shows the fraction of pipelines where the candidate repair hurts performance. Janus repairs hurt 4% – 39% of pipelines, less than

[6]Recall tree edit distance scales as a function of *both* tree sizes, so effective pruning can be even faster than random sampling if the trees sampled are smaller in size.



(a) Distribution of types for LSRs lifted from edit operations.

(b) Distribution of types for LSRs after Janus has summarized them.

Figure 5-4: Janus mines a total of 40,232 local structural rules, which are then summarized to a rule corpus of 2,939 local structural rules using their LSR keys. Janus's summarized rule corpus presents a more balanced mix of rule types compared to the raw collection of LSRs.

baseline approaches in 6 of our 9 datasets. We compared Janus's performance with the next best baseline, *Meta-Learning*, using a McNemar paired test over repair outcomes (i.e., was a pipeline successfully repaired by either, both, or just one of the systems) and find that there is a statistically significant difference in their performance (95 test statistic, p-value <0.01).

Figure 5-5 shows the ECDF for the score change in pipelines that improved (Figure 5-5a) or were hurt (Figure 5-5b) as a result of a candidate repair. We find that when candidate repairs hurt the performance of a pipeline the decrease induced by Janus is less than that under other approaches. When the pipeline score is improved, the improvements produced by Janus are comparable to those produced by baselines *Janus-Random* and *Random-Mutation*, but less than those obtained by *Meta-Learning*. But when both Janus and *Meta-Learning* produce an improvement on the same input pipeline, we find that a Wilcoxon Signed Rank test (1128 test statistic, p-value 1.0) did not show a statistically significant difference in paired scores. The mean repair time for all approaches is comparable at approximately 3 seconds in all datasets. All original pipelines and their corresponding Janus candidate repairs (along with score information) are available in JSON format.[7]

[7] `https://github.com/josepablocam/janus-public/blob/main/janus-repairs-formatted.json`

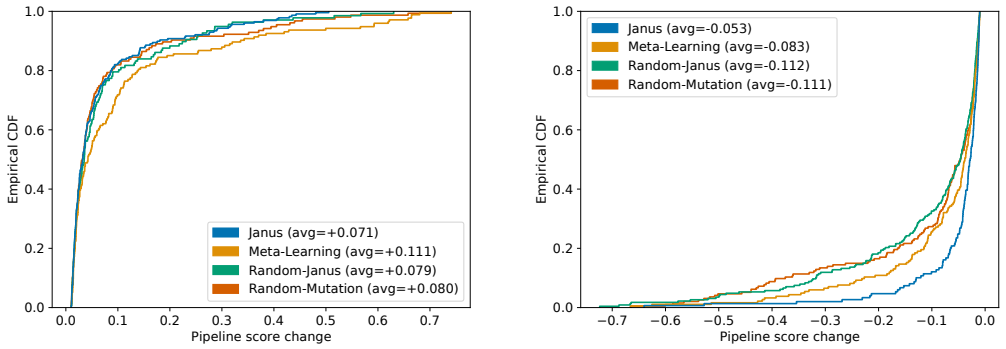
dataset	Janus	Meta-Learning	Random-Janus	Random-Mutation
Hill_Valley_with_noise	0.19 (0.12-0.26)	**0.30 (0.21-0.37)**	0.13 (0.07-0.18)	0.16 (0.09-0.22)
Hill_Valley_without_noise	0.16 (0.10-0.22)	**0.20 (0.13-0.26)**	0.15 (0.09-0.21)	0.08 (0.03-0.12)
breast-cancer-wisconsin	**0.20 (0.14-0.26)**	0.11 (0.06-0.15)	0.04 (0.01-0.07)	0.14 (0.08-0.19)
car-evaluation	**0.27 (0.20-0.35)**	0.16 (0.10-0.22)	0.25 (0.18-0.31)	0.21 (0.14-0.28)
glass	**0.42 (0.34-0.50)**	0.26 (0.19-0.34)	0.28 (0.20-0.35)	0.38 (0.29-0.46)
ionosphere	**0.26 (0.18-0.33)**	0.21 (0.14-0.28)	0.08 (0.03-0.12)	0.15 (0.09-0.20)
spambase	**0.19 (0.12-0.25)**	0.14 (0.07-0.19)	0.07 (0.02-0.11)	0.09 (0.04-0.14)
wine-quality-red	**0.25 (0.17-0.32)**	0.14 (0.09-0.20)	0.17 (0.11-0.23)	0.14 (0.08-0.20)
wine-quality-white	**0.36 (0.28-0.44)**	0.25 (0.18-0.31)	0.21 (0.13-0.27)	0.21 (0.13-0.28)

(a) Fraction of input pipelines improved

dataset	Janus	Meta-Learning	Random-Janus	Random-Mutation
Hill_Valley_with_noise	**0.11 (0.06-0.16)**	0.16 (0.10-0.21)	0.20 (0.13-0.28)	0.20 (0.13-0.27)
Hill_Valley_without_noise	**0.04 (0.01-0.07)**	0.17 (0.11-0.23)	0.11 (0.06-0.16)	0.21 (0.14-0.27)
breast-cancer-wisconsin	**0.04 (0.01-0.07)**	0.18 (0.12-0.24)	0.15 (0.09-0.21)	0.14 (0.08-0.19)
car-evaluation	0.35 (0.27-0.43)	**0.21 (0.14-0.27)**	0.45 (0.36-0.54)	0.32 (0.24-0.40)
glass	0.39 (0.31-0.47)	0.37 (0.29-0.45)	0.38 (0.30-0.46)	**0.34 (0.26-0.41)**
ionosphere	**0.20 (0.13-0.26)**	0.28 (0.21-0.35)	0.21 (0.14-0.28)	0.26 (0.18-0.33)
spambase	**0.04 (0.01-0.07)**	0.09 (0.04-0.14)	0.14 (0.07-0.19)	0.05 (0.01-0.08)
wine-quality-red	0.21 (0.14-0.28)	**0.20 (0.13-0.26)**	0.29 (0.22-0.36)	0.23 (0.16-0.30)
wine-quality-white	**0.11 (0.06-0.16)**	0.19 (0.12-0.25)	0.36 (0.28-0.43)	0.20 (0.13-0.26)

(b) Fraction of input pipelines hurt

Table 5.1: Fraction of input pipelines improved or hurt by a candidate repair, along with 95% confidence intervals in parentheses. Janus improves performance of 16%–42% of pipelines on average, outperforming other approaches in 7 of our 9 datasets. Conversely, Janus candidate repairs degrade performance of 4%–39% of pipelines, less than other approaches in 6 of our 9 datasets.



(a) ECDF over score changes for pipeline improvements

(b) ECDF over score changes for pipeline degradations

Figure 5-5: Empirical Cumulative Distribution Functions (ECDFs) over pipeline score changes after candidate repairs. When candidate repairs hurt performance, Janus results in smaller degradations than other approaches. Janus score improvements are comparable to *Janus-Random* and *Random-Mutation* but less than those of *Meta-Learning*.

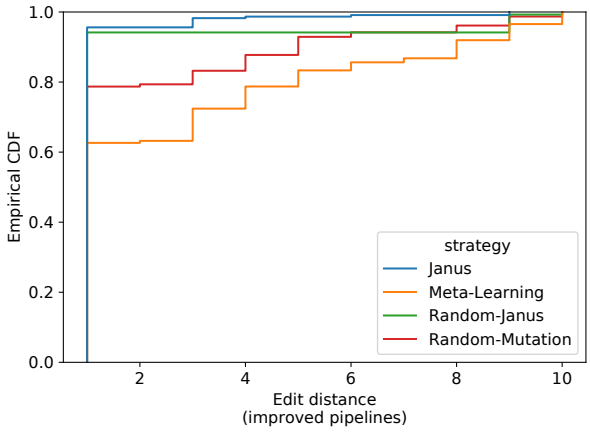


Figure 5-6: Approaches that use Janus-mined rules (Janus and *Janus-Random*) produce slightly closer repairs compared to *Meta-Learning* and *Random-Mutation*.

5.6.3 Repair distance

Figure 5-6 shows the ECDF of tree edit distance (with respect to the original pipelines) for pipelines improved by the corresponding system. We see that repairs produced using Janus-mined rules (i.e., Janus and *Janus-Random*) tend to produce closer repairs compared to *Random-Mutation* and *Meta-Learning*. When generating candidate repairs, Janus mostly applies hyperparameter update rules (82%), followed by hyperparameter removal rules (11.6%), and component insert/remove rules (3.7% and 2.1% respectively).[8] Hyperparameter-related rules produce an edit distance of 1, which drives Janus repair distances down.

5.6.4 Sensitivity to Pipeline Corpus

Pipeline repairs are only as effective as the rules that can be mined from the pipeline corpus. To evaluate the sensitivity of Janus to the underlying pipeline corpus, we conduct an experiment in which we follow the methodology in Section 5.5 to create the pipeline corpus but instead of using TPOT to generate pipelines, we use a random pipeline search process. In this method, we first randomly sample the depth of the pipeline from a $U(1,k)$ distribution (where we set $k=4$), then iteratively sample uniformly at random a component and a hyperparameter configuration for that component from the pre-configured TPOT

[8] Component update rules were a negligible fraction.

dataset	Janus	Meta-Learning	Random-Janus	Random-Mutation
Hill_Valley_with_noise	0.25 (0.18-0.33)	**0.28 (0.19-0.35)**	0.27 (0.18-0.35)	0.27 (0.18-0.34)
Hill_Valley_without_noise	0.24 (0.16-0.31)	**0.27 (0.18-0.34)**	0.23 (0.16-0.30)	0.20 (0.13-0.28)
breast-cancer-wisconsin	0.26 (0.18-0.33)	**0.36 (0.27-0.44)**	0.29 (0.20-0.37)	0.25 (0.18-0.32)
car-evaluation	0.33 (0.24-0.41)	0.40 (0.31-0.48)	**0.41 (0.33-0.50)**	0.34 (0.26-0.42)
glass	0.43 (0.35-0.51)	**0.48 (0.38-0.57)**	0.24 (0.15-0.31)	0.32 (0.24-0.40)
ionosphere	**0.42 (0.34-0.51)**	0.33 (0.24-0.41)	0.36 (0.28-0.45)	0.33 (0.24-0.41)
spambase	0.29 (0.20-0.37)	**0.32 (0.23-0.41)**	0.30 (0.22-0.38)	0.20 (0.13-0.27)
wine-quality-red	0.33 (0.24-0.42)	**0.35 (0.26-0.44)**	0.29 (0.22-0.38)	0.27 (0.18-0.35)
wine-quality-white	**0.33 (0.25-0.42)**	0.30 (0.21-0.37)	0.23 (0.14-0.30)	0.30 (0.21-0.38)

Table 5.2: Fraction of pipelines improved when we replace our corpus of pipelines with one built using random search. We report 95% confidence intervals in parentheses. Janus still repairs 24%–43% of pipelines but only outperforms in 2 of the 9 datasets underscoring the importance of the underlying pipeline corpus.

search space (`tpot.config.classifier_config_dict`).[9] The rest of the setup is identical. We collect 19,551 pipelines, extract 39,496 LSRs, and compile a summarized rule corpus of 4,926 rules. Our test pipelines are similarly drawn from this new pipeline corpus.

Table 5.2 shows that in this setting Janus still improves between 24%–43% of pipelines, but now only outperforms in 2 of the 9 datasets. This highlights the importance of the underlying pipeline corpus.

5.7 Threats to Validity

Our experiments rely on a corpus of pipelines produced by an automated tool (TPOT [82]). It is possible that the effectiveness of Janus will vary based on the underlying distribution of pipelines in the corpus. Our experimental results on corpus sensitivity show that a different corpus can impact Janus's ability to outperform but the rules mined still repair a significant fraction of input pipelines. This risk can be further mitigated by increasing the size and sophistication of the underlying pipeline corpus.

Our experiments restrict candidate repairs to be within a $d=10$ tree edit distance of the original input pipeline for all approaches compared. Increasingly far away candidate repairs may display different performance characteristics, but the goal of Janus is not to produce the single largest performance increase but rather increase performance by

[9]This method loosely reduces to a version of TPOT that does not use any genetic programming and produces only sequential pipelines.

producing a *nearby* pipeline.

Janus relies on a simple key-based approach to rule abstraction. It is possible that other abstraction procedures, for example deduction-based techniques such as anti-unification [69], may yield rules with different performance. In particular, pipelines that require edits across many components may not be amenable to improvement with Janus, which lifts individual edit operations to repair rules. However, this risk is mitigated as in practice many real pipelines implemented in scikit-learn (Janus' target library) have less then 4 components [92].

As our experiments show, candidate repairs from both Janus and baseline approaches can also degrade pipeline performance. To mitigate this risk, repair systems could validate the performance of candidate repairs on held-out data to obtain a performance estimate and determine if it improves meaningfully over their input pipeline.

5.8 Conclusion

We framed the task of improving an *existing* machine learning pipeline's performance through a small transformation as an analogue to automated program repair. In this setting, the original pipeline, the pipeline change, and a procedure for automatically mining and applying such modifications as analogues to bug, patch, and program repair. We present Janus, a system that mines repair rules for machine learning pipelines by analyzing a large corpus of pipelines. We show that Janus can use these rules to improve between 16%–42% of our test pipelines, outperforming baseline approaches in 7 of our 9 test datasets.

Chapter 6

Related Work

We focus our discussion of related work on automated machine learning (Section 6.1) and systems that mine information from software artifacts (Section 6.2).

6.1 Automated Machine Learning Systems

As introduced in Section 2.3, automated machine learning is traditionally framed as an optimization problem, where the system, given a compute time budget and an input dataset, is tasked with generating (or selecting from a portfolio) the pipeline(s) with the lowest test error. A wide variety of systems and techniques have been developed to tackle this challenge. We focus on presenting an overview of this work and placing our thesis contributions in context.

Random search: Despite its simplicity, random search has been effectively employed within the context of automated machine learning. When using random search, a system may randomly propose hyperparameter configurations, pipeline components, or complete pipeline definitions. Bergstra and Bengio [12] showed that certain models may have a large number of hyperparameters but only a subset of these impact performance (i.e. the dimensionality is effectively lower), and in such settings random search can approach or improve over the performance obtained by a combination of manual and grid-based search.

The approach used in AMS to mine hyperparameter search spaces from code examples is based on a related notion of "low effective dimensionality". In particular, our counting-

based approach to collecting popular hyperparameters from source calls is based on the insight that users tend to tune a small subset of the hyperparameters available for a given library component. Additionally, the current implementation of AMS includes random search as a procedure to sample pipelines from an automatically augmented search space.

Ensembling: The effectiveness of random search can also be improved by combining resulting pipelines using ensembling [29]. Caruana et al [21] showed that ensemble selection, a greedy forward sampling with replacement of existing models, improved performance. Chen et al's Autostacker [23] employs layered stacking to produce an ensemble of randomly generated model configurations. Similarly, the commercial system H2o AutoML [71] employs a stacking ensemble to combine pipelines generated through random search.

While our evaluation of AL focuses on the predictive performance of individual pipelines, our implementation also allows users to ensemble all pipelines generated for classification tasks by voting for each observation's most frequently predicted label. Stacking the predictions produced by AL-generated pipelines and then feeding these to a separate meta-predictor, as done by the H2o AutoML system, might improve performance further.

Bayesian optimization and Sequential Model-based Algorithm Configuration: Snoek et al [108] showed that Bayesian optimization (BO) can tune hyperparameters more effectively than random search. Challenges arise in scaling Gaussian-process-based BO to high dimensions, which is particularly important for automated machine learning. Bergstra et al [11] introduced the use of tree-structured Parzen estimators, which can represent structured hyperparameter spaces (i.e. where one hyperparameter value depends on another), and scales well to higher dimensions. Based on these ideas, Bergstra et al developed Hyperopt [13]. Hyperopt users can manually define a structured search space, along with prior probability distributions over the space. Hyperopt can use these prior distributions (along with experiment history) to propose and evaluate increasingly better hyperparameter configurations.

Hutter et al [57] introduced SMAC, which extends the general BO framework to handle many of the challenges associated with automated machine learning (and more broadly, "algorithm configuration"). In particular, SMAC handles a mix of numeric and non-numeric

hyperparameters, employs random forest regressors as the surrogate model used in the acquisition function, and uses a racing algorithm, ROAR, to decide if a new configuration actually outperforms the incumbent configuration. Based on SMAC, Auto-Weka [110, 66] was one of the first complete automated machine learning systems, performing not only hyperparameter optimization but also component selection and composition.

More recently, Autosklearn [41] uses SMAC [57] to explore pipelines composed of feature/data preprocessors and a classifier or regressor. The system searches over a pre-selected set of classes from scikit-learn and XGBoost, and a manually identified subset of hyper-parameters for each selected component. Autosklearn uses SMAC to predict the performance of each configuration (selection of components and corresponding hyper-parameter settings) and explores different parts of the search space to gain new information. Autosklearn also employs meta-learning by training a series of pipelines offline and then using these pipelines to initialize the search based on dataset characteristics. Additionally, rather than return a single pipeline, autosklearn creates an ensemble from generated candidates. More recently, autosklearn 2.0 [40] extends autosklearn to incorporate early stopping, a larger search space, and a bandit-based approach for allocating evaluation time among candidate pipelines.

The design of AL focuses on the challenges of component selection and composition, and as a result generated pipelines use default hyperparameters with no additional tuning. However, in Section 3.7.2 we provide a detailed proposal to extend AL with hyperparameter optimization by integrating a tuning phase after the initial set of pipelines is generated. This tuner could be instantiated to use Bayesian optimization.

Similarly, while the current implementation of Janus returns the first transformed pipeline that satisfies the user-provided edit distance constraint and executes without runtime exceptions, the system can be easily extended to adopt a traditional generate-and-validate approach, where multiple transformed trees are empirically evaluated and their performance results inform subsequent transformation attempts. In particular, we can generate multiple pipeline candidates using transformations mined by Janus and then (repeatedly) choose a concrete candidate to evaluate based on Bayesian optimization (e.g. maximizing expected improvement predicted by a surrogate model).

A key advantage of AL, in contrast to systems such as Autosklearn and Auto-Weka, is that the system designer does not need to manually identify the components to be included in the system's search space. As we show in our evaluation, by identifying these components from example programs, AL results in a more flexible system that can generalize to new datasets where other systems fail to execute.

Multi-Armed Bandits: Multi-armed bandits (MABs) [17] are another popular optimization tool in the context of automated machine learning as they provide an effective optimization technique for exploring a space of discrete choices with noisy reward functions. Autobazaar [107] employs a combination of MAB optimization and BO to produce pipelines, where the former proposes pipeline templates, which are then tuned by the latter. Alpine Meadow [105] uses a similar set of optimization techniques to produce pipelines, framing the choice of an abstract pipeline (i.e. without tuned hyperparameters) as a MAB. For both Autobazaar and Alpine Meadow, system designers need to populate the set of templates that instantiate the MAB. In particular, Autobazaar employs hand-curated templates, while Alpine Meadow relies on a collection of pipelines evaluated offline to seed their templates. In both cases, AL's approach to generating pipelines could complement these MAB-based approaches. In particular, AL-generated pipelines could be used as pipeline templates to seed a MAB.

We could also incorporate MABs into other systems presented in this thesis. For example, we might group the rules mined by Janus based on their respective target node labels and treat these groups as arms in a MAB. Doing so would hypothesize that modifying certain target nodes is more likely to improve performance. Within each such group, we could continue to prioritize individual transforms based on their joint probability, as done in the current Janus implementation.

Genetic Programming: Evolutionary search methods have also been used to optimize expensive, black-box functions [62]. Olson et al [82] formulate the task of generating tree-structured pipelines as a genetic programming (GP) problem. Recent work by Le et al [70] extends TPOT with a domain-specific feature selection component and pipeline templates (abstract pipelines, where components can be restricted to discrete sets) to

scale TPOT to larger datasets. Similarly, De Sá et al's RECIPE [27] employs genetic programming over a grammar that defines valid pipelines to reduce the number of invalid candidates generated during the GP search.

We employ GP in our evaluation of search spaces generated by AMS. In particular, we use TPOT as one of the two search methods in our experiments. Our results show that AMS can effectively generate a search space capable of producing high performing pipelines when fed to a GP-based AutoML tool. Relatedly, we compare Janus to a random mutation baseline and find that using Janus-mined transformations on average produced a higher fraction of pipeline improvements. It may be possible that integrating Janus transformations into a GP search, as a replacement to the traditional random mutation operation, could produce higher performing pipelines or reduce search times.

Reinforcement Learning: Modeling the task of pipeline generation as a sequence of actions, where components are added, removed, or replaced makes the use of reinforcement learning appealing. Drori et al [32, 31] used reinforcement learning as the underlying optimization technique in their AlphaD3M automated machine learning system. Sun et al [109] developed ReinBO, which uses reinforcement learning to sequentially generate pipelines without concrete hyperparameters and then applies BO to tune the hyperparameters available, propagating back performance information to both the BO surrogate model and the reinforcement learning system.

While Janus extracts pipeline transformations based on insertion, update, and deletion operations over the pipelines' tree representations, it does not model applying these transformations as a reinforcement learning problem. Janus instead performs a greedy search, returning the first transformed pipeline to satisfy a distance constraint and to execute without runtime exceptions. In particular, while the implementation of Janus allows for multiple edits to be composed (through sequential application), doing so often yields pipelines that are beyond the distance constraint, and thus not the desired output for the use case underpinning the design of Janus. However, it possible that the transformations mined by Janus could be complementary to existing reinforcement learning approaches. One possible way to integrate these approaches would be to scope the actions available to the agent to be the transformations mined by Janus.

Meta-learning: Meta-learning, where systems exploit performance information from prior tasks, has also been used successfully in automated machine learning. Feurer et al [41, 40] collect performance information (offline) for a large portfolio of pipelines and use high-level dataset characteristics to choose an initial set of pipeline candidates for a new input dataset. Task, data, and API descriptions can also be incorporated into a meta-learning approach. Drori et al [33] embed these natural language descriptions and use them to perform zero-shot pipeline suggestions, drawing from a portfolio of previously evaluated pipelines. Perrone et al [90] learn bounding boxes for the candidate values proposed in BO for numeric hyperparameters by running BO offline and learning from the history of optimal values. Yang et al's OBOE [121] and TensorOBOE [122] evaluate the performance of a large portfolio of pipelines, in an offline process, and then in an online process use active learning in conjunction with matrix/tensor completion techniques to evaluate the performance of a small number of pipelines on a new dataset and infer the performance of the broader portfolio.

In general, mining software artifacts may be viewed as software-oriented form of meta-learning. Much like meta-learning exploits information, for example predictive performance, associated with previous datasets or pipelines, mining software artifacts exploits information associated with previous programs or development by-products (e.g. documentation). AL, for example, relies on collecting dynamic traces from a corpus of programs that implement their own machine learning pipelines. In contrast to systems such as Autosklearn, AL does not collect performance information from these pipelines, but rather takes an unsupervised approach and models pipeline probability. Similarly to Perrone et al, AMS collects hyperparameter search spaces, but rather than produce these search spaces by running searches on datasets offline and collecting optimal values, AMS collects the most frequently observed values from existing users' source code.

Domain Specific Languages: Recent work has also explored the use of domain specific languages as a way to formalize pipeline search spaces and organize the variety of optimization techniques available. LALE [54] is a domain specific language that allows developers to define machine learning pipelines and optimization choices, producing a structured definition for an automated procedure's candidate search space. Our system AMS, in particular, focuses on the related problem of automatically generating a search

space from a user-chosen set of API components. AMS could easily be extended to output its search spaces as LALE programs, which would allow users to take advantage of the different search tools that have been incorporated into LALE's backend.

Importantly, the systems presented in this thesis are not focused on any one particular learning or optimization technique. For example, in AL, we model pipeline component selection/composition by training linear models to predict the next component in a partially completed pipeline. In AMS, we rely on information retrieval techniques. In particular, AMS uses a popular textual similarity metric (BM25) to identify related components from their API documentation and a probabilistic measure of association (normalized pointwise mutual information) to identify components that co-occur in user scripts. In Janus, we rank candidate tree transforms and their target location using the joint probability (computed over the corpus of pipelines collected) of a local structural rule key and a target node's label.

6.2 Mining Software Artifacts

The techniques underlying the systems presented in this thesis share conceptual underpinnings with existing research in programming languages and software engineering. In particular, we focus on discussing existing work that makes use of users' source code, program executions, program differences, and documentation to tackle problems such as program synthesis, program verification, program repair, and API usage.

Source code: Mining source code can be an effective way of identifying repeated developer patterns. For example, Shin et al's PATOIS [106] identifies repeated code patterns that are associated with idioms, abstracts these idioms, and incorporates the abstracted idioms into a neural program synthesizer to generate programs that satisfy an input-output-based specification. By extending the synthesizer's domain specific language to incorporate idioms, the search for programs can incorporate both a high- and low-level exploration strategy to generate candidates.

AL, in particular, can be viewed as a component-based synthesis system, which is given a soft specification: pipelines must maximize a performance metric. In its current implementation, AL's pipeline search builds up pipelines incrementally, adding components

based on their predicted probability, given a partial pipeline and the dataset state. AL could be extended to incorporate a set of higher-level components, such as frequently recurring subsequences of components observed in user programs. Incorporating these higher-level components could accelerate the search by removing the need for repeated extension steps.

Prior work, such as [49, 72, 18, 56], has investigated the use of a parallel corpus of natural language descriptions (mined from documentation strings) and source code to enable natural-language-based code search. AMS, like these systems, exploits the availability of natural language descriptions for target API components. However, AMS does not pair descriptions with user code fragments. Instead, AMS uses the natural language descriptions to identify API classes that are related. An aligned-corpus of natural language and source code could be used to extend the way in which AMS identifies complementary components. In particular, given natural language descriptions of a dataset or task and associated source code written by users for that predictive task, AMS could allow new users to restrict the subset of programs used to mine complementary components to those that have a description most closely associated with their own. This is similar to the use of dataset and task descriptions by Drori et al [33], who create embeddings from natural language descriptions of the data and task to perform zero-shot pipeline retrieval. In contrast, AMS would use components retrieved to extend a search space and then sample pipelines by composing components from that space.

Head et al's CodeScoop [53] mines specific user scripts, rather than a large collection of programs, to automatically produce minimal working examples of program functionality which can be used for debugging or explanatory purposes. To extract the minimal version of the desired functionality, CodeScoop relies on a combination of static and dynamic analysis of the underlying source code. For all three systems presented in this thesis, we rely on mining large collections of artifacts rather than individual ones. For example, AL by definition requires multiple example programs in order to compute the probability of particular pipeline extensions. Similarly, AMS relies on having a corpus of example programs that is large enough to identify complementary components through relative co-occurrence and popular hyperparameters and their values through their frequency.

Program executions: Extracting (and summarizing) information from program executions can be used to address different development challenges. A notable example of

summarizing information across program executions is Daikon [36], which mines program invariants from dynamic executions. Such dynamic invariants have been successfully used to model and identify API misusage [68], localize errors [2], and automatically repair buggy programs [30]. Importantly, dynamic program invariants reflect key program properties, such as the relationship between different variables at different points in the program's execution. In AL's design meta-features play a similar role, summarizing the properties of the input dataset before different components are called in a pipeline. In contrast to invariants, AL's meta-features consist of a set of pre-defined functions that compute key statistics, such as the number of columns/rows or percent of missing values in a dataset. Furthermore, AL does not relate meta-features for a dataset before and after a component has been applied. A possible use of dynamic invariants may indeed be to extract relationships between meta-features before and after a data transformation component has been applied. For example, given a meta-feature that computes the fraction of missing values and a component that performs data imputation, we would likely extract an invariant that states that applying such a component makes the associated meta-feature zero. These invariants could be used to extend AL's pipeline generation, for example prioritizing candidate pipelines that satisfy a larger number of invariants mined from user-written pipelines.

DemoMatch [125] relies on user demonstrations of an application to extract a program trace, a sub-trace of which is associated with a desired application feature. The partial trace collected by DemoMatch can then be used to search an existing repository of traces, from which the system can synthesize example code (including key setup steps) exercising the user's target functionality. In contrast to DemoMatch, AL does not use the dynamic traces collected to produce example code that exercises existing functionality, nor does AL search through a database of existing program traces to retrieve relevant sets of traces. AL uses the dynamic traces collected to build up models for partial pipeline completion, which it can then use to generate pipelines based on their predicted probability. Modeling pipeline probability, and performing an incremental search for pipeline candidates, means AL is not restricted to returning concrete pipelines observed (nor subsequences of observed pipelines). Additionally, a developer who makes use of AL to generate a set of pipelines does not need to demonstrate functionality in an existing application but instead provides only their input tabular dataset.

Program differences: Automated program repair literature has a long history of mining paired program versions and developer patches. For example, Long and Rinard developed Prophet [76], an automated repair system for C that builds a model for patch correctness by mining code features found in developer patches. Long et al's Genesis [74] mines and generalizes code transformations from human patches that can be used to produce candidate repairs in an automated system. Rolim et al's REFAZER [99] and REVISAR [100] framed the task of learning code transforms from revisions as a program synthesis task and show that these synthesized programs can be used to effectively perform repeated program transformations. Bader et al's Getafix [7] introduced a hierarchical clustering approach for organizing mined repair patterns, which powered an industrially-deployed automated repair system.

Similarly to these systems, Janus relies on extracting pipeline transformations from paired pipelines, one of which is lower performing than the other on the same dataset. However, in contrast to this body of work, Janus is focused on mining and transforming machine learning pipelines. Critically, different machine learning pipelines may result in different predictive performance, and as such the outcome for transforming pipelines is continuous rather than the binary outcome (pass/fail) associated with traditional program repair. This difference is reflected in the design of Janus. For example, the heuristic summarization of local structural rules in Janus relies on keeping the concrete rule with the highest performance differential for each rule key. Extending Janus to make use of other rule abstraction procedures may be possible, however, doing so would require that we incorporate notions of predictive performance (as a continuous outcome) into that summarization.

Documentation: Pandita et al [87] developed TMAP, a system that uses text mining of API method descriptions to identify possible mappings between two APIs, facilitating developer porting of a program to a new API. Ni et al [80] exploit documentation to develop SOAR, a synthesis-based approach to rewriting an existing data science pipeline using a new API. To identify similar functions in the source and target APIs Ni et al compute a textual similarity representation, using both TF-IDF and a TF-IDF-weighted average of GloVe embeddings. In addition to mining API descriptions, SOAR also extracts runtime error messages from failed candidate programs and uses these to further restrict the search space of pipeline rewrites.

Similar to both TMAP and SOAR, AMS mines the documentation available for our target API (scikit-learn). In particular, AMS uses documentation strings to automatically identify API components that may be functionally related, and which can be integrated into a search space. Similar to TMAP and SOAR, AMS relies on a lexical notion of textual similarity, computing BM25 over API documentation strings. However, in contrast to TMAP and SOAR, AMS focuses on comparing documentation for components from the same API, rather than targeting cross-API mappings, and similar components identified are added to an AutoML search space, rather than used to rewrite an existing program.

In our evaluation of AMS, we compare the precision associated with retrieving functionally-related components using BM25 as a similarity metric compared to a similarity metric that incorporates pre-trained neural embeddings. Our results show that, in the context of AMS, both of these approaches result in comparable precision. Similarly, SOAR's evaluation shows that a version of their system that uses just TF-IDF to compute API-mappings performs favorably compared to a version that incorporates GloVe embeddings. In both AMS and SOAR, it seems that a factor in the lack of improvement from the use of distributed term embeddings is that they were pre-trained on a corpus of natural language from a domain that is not code-specific (and in particular, not data science or machine learning related).

Unlike SOAR, none of the systems presented in this thesis make explicit use of error messages resulting from invalid pipeline definitions. However, AL's candidate generation and the repair process in Janus (as well as searches carried out on search spaces produced by AMS) can result in invalid pipelines that raise runtime exceptions. Indeed, the implementations of most AutoML systems are riddled with exception handling blocks to address failed pipeline candidates encountered during their search procedures. Integrating error messages into the candidate pipeline generation, for example identifying components that repeatedly fail based on exception messages produced and pruning these from the search space, could reduce the number of failed pipeline definitions and reduce search times as a consequence.

Chapter 7

Conclusion

AutoML promises to make machine learning more accessible, efficient, and impactful. However, the current state of AutoML can be (roughly) characterized as system designers spending significant amounts of time devising increasingly sophisticated search techniques. While these techniques *do* result in better performance on a variety of benchmark datasets, research is increasingly showing that AutoML users also want systems that provide transparency, interpretability, customizability, and useability [119, 118, 120] – needs, that in my view, will have to be addressed for widespread adoption of AutoML to become a reality. Furthermore, while advances in related areas like neural architecture search are exciting from a research perspective, it is my view that AutoML focused on classical machine learning pipelines holds the most potential for near-term widespread impact – repurposing Dunkin Donut's slogan "America (still) runs on spreadsheets and logistic regression".[1]

To realize this vision of AutoML we need to shift towards a user- (and developer-) centric view. And a key for facilitating this shift is to make use of information available about *existing* developer and user behaviors and resources. This thesis presents evidence that this information can be effectively mined from software artifacts, which can encode current development practices, provide information for user customization, and provide information for software evolution. Collectively, these systems can successfully address design questions such as *what components do developers actually use when they manually write their own pipelines?*, *how do we enable search space customization by non-expert users?*, and *what*

[1] Dunkin Donut's original slogan is "America runs on Dunkin".

changes could be made to an existing pipeline to improve its predictive performance?

Research in programming languages and software engineering has provided developers with a powerful toolkit that can tackle tasks of program verification, repair, retrieval, synthesis, optimization, and more. An analogous toolkit for machine learning pipelines, which are increasingly implemented by individuals who are not professional software developers, can have an outsized impact. Mining artifacts for traditional programs has accelerated and improved these repair [74], verification [14], retrieval [49], and synthesis [106] systems. Adapting these insights to automating machine learning pipeline development can bring about exciting opportunities such as generating pipelines from natural language, automated testing and repair of defects in pipelines, and automated pipeline rewriting across environments or computational targets (e.g. different architectures, languages).

Bibliography

[1] Uci: Solar flare data set, 2017.

[2] Rui Abreu, Alberto González, Peter Zoeteweij, and Arjan JC van Gemund. Automatic software fault localization using generic program invariants. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 712–717, 2008.

[3] Shawkat Ali and Kate A Smith-Miles. A meta-learning approach to automatic kernel selection for support vector machines. *Neurocomputing*, 70(1):173–186, 2006.

[4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[5] Limor Appelbaum, José P Cambronero, Jennifer P Stevens, Steven Horng, Karla Pollick, George Silva, Sebastien Haneuse, Gail Piatkowski, Nordine Benhaga, Stacey Duey, et al. Development and validation of a pancreatic cancer risk model for the general population using electronic health records: An observational study. *European Journal of Cancer*, 143:19–30, 2021.

[6] Autosklearn. Github repository issue 292, 2017.

[7] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[8] Alejandro Correa Bahnsen, Djamila Aouada, Aleksandar Stojanovic, and Björn Ottersten. Feature engineering strategies for credit card fraud detection. *Expert Systems with Applications*, 51:134–142, 2016.

[9] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.

[10] Iz Beltagy, Arman Cohan, and Kyle Lo. Scibert: Pretrained contextualized embeddings for scientific text. *arXiv preprint arXiv:1903.10676*, 2019.

[11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation, 2011.

[12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[13] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[14] Dirk Beyer. A data set of program invariants and error paths. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 111–115. IEEE, 2019.

[15] Gerlof Bouma. Normalized (pointwise) mutual information in collocation extraction. *Proceedings of GSCL*, pages 31–40, 2009.

[16] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[17] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721*, 2012.

[18] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974, 2019.

[19] José P. Cambronero, Jürgen Cito, and Martin C. Rinard. Ams: Generating automl search spaces from weak specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 763–774, New York, NY, USA, 2020. Association for Computing Machinery.

[20] José P. Cambronero and Martin C. Rinard. Al: Autogenerating supervised learning programs. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[21] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.

[22] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[23] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. Autostacker: A compositional evolutionary learning system. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 402–409, 2018.

[24] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[25] François Chollet et al. Keras. `https://keras.io`, 2015.

[26] James Michael Curran, Tacha Natalie Hicks Champod, and John S Buckleton. *Forensic interpretation of glass evidence*. CRC Press, 2000.

[27] Alex GC de Sá, Walter José GS Pinto, Luiz Otavio VB Oliveira, and Gisele L Pappa. Recipe: A grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming*, pages 246–261. Springer, 2017.

[28] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):1–19, 2009.

[29] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, pages 1–15, Berlin, Heidelberg, 2000. Springer-Verlag.

[30] Zhen Yu Ding, Yiwei Lyu, Christopher S. Timperley, and Claire Le Goues. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *Proceedings of the 6th International Workshop on Genetic Improvement*, GI '19, pages 2–9. IEEE Press, 2019.

[31] Iddo Drori, Yamuna Krishnamurthy, Raoni Lourenco, Remi Rampin, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar. *arXiv preprint arXiv:1905.10345*, 2019.

[32] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni Lourenço, Jorge One, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Alphad3m: Machine learning pipeline synthesis. In *AutoML Workshop at ICML*, 2018.

[33] Iddo Drori, Lu Liu, Yi Nian, Sharath C Koorathota, Jie S Li, Antonio Khalil Moretti, Juliana Freire, and Madeleine Udell. Automl using metadata language embeddings. *arXiv preprint arXiv:1910.03698*, 2019.

[34] Mikael Elinder and Oscar Erixson. Gender, social norms, and survival in maritime disasters. *Proceedings of the National Academy of Sciences*, 109(33):13220–13224, 2012.

[35] Sophie Emerson, Ruairí Kennedy, Luke O'Shea, and John O'Brien. Trends and applications of machine learning in quantitative finance. In *8th international conference on economics and finance research (ICEFR 2019)*, 2019.

[36] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. Association for Computing Machinery.

[37] Ian W Evett and Ernest J Spiehler. Rule induction in forensic science. In *Knowledge Based Systems*, pages 152–160. Halsted Press, 1989.

[38] Fabio Fabris and Alex A Freitas. Analysing the overfit of the auto-sklearn automated machine learning tool. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 508–520. Springer, 2019.

[39] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.

[40] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074*, 2020.

[41] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[42] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[43] Akinori Fujino, Hideki Isozaki, and Jun Suzuki. Multi-label text categorization with model combination based on f1-score maximization. In *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-II*, 2008.

[44] Christophe Giraud-Carrier, Ricardo Vilalta, and Pavel Brazdil. Introduction to the special issue on meta-learning. *Machine learning*, 54(3):187–193, 2004.

[45] GitHub. Github, 2021.

[46] Google. Kaggle website, 2017.

[47] Google. Kaggle website (time limits), 2017.

[48] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[49] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[50] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, et al. A brief review of the chalearn automl challenge: any-time any-dataset learning without human intervention. In *Workshop on Automatic Machine Learning*, pages 21–30, 2016.

[51] Satoshi Hara and Kohei Hayashi. Making tree ensembles interpretable: A bayesian model selection approach. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 77–85, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR.

[52] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[53] Andrew Head, Elena L Glassman, Björn Hartmann, and Marti A Hearst. Interactive extraction of examples from existing code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[54] Martin Hirzel, Kiran Kate, Avraham Shinnar, Subhrajit Roy, and Parikshit Ram. Type-driven automated learning with lale. *arXiv preprint arXiv:1906.03957*, 2019.

[55] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[56] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[57] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[58] F. Ishikawa and N. Yoshioka. How do engineers perceive difficulties in engineering of machine-learning systems? - questionnaire survey. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 2–9, 2019.

[59] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[60] Kaggle. Titanic: Machine learning from disaster (start here! predict survival on the titanic and get familiar with ml basics), 2015.

[61] Kaggle. Meta-kaggle, 2017.

[62] John Karro, Greg Kochanski, and Daniel Golovin. Black box optimization via a bayesian-optimized genetic algorithm. In *Proc. OPTML 2017: 10th NIPS Workshop Optim. Mach. Learn.*, 2017.

[63] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, pages 1265–1276, 2017.

[64] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[65] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038, 2016.

[66] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 17:1–5, 2016.

[67] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020.

[68] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 178–189, New York, NY, USA, 2014. Association for Computing Machinery.

[69] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *Journal of Automated Reasoning*, 52(2):155–190, 2014.

[70] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, 06 2019.

[71] Erin LeDell and S Poirier. H2o automl: Scalable automatic machine learning. In *7th ICML workshop on automated machine learning*, 2020.

[72] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. Neural query expansion for code search. In *Proceedings of the 3rd acm sigplan international workshop on machine learning and programming languages*, pages 29–37, 2019.

[73] Steve Lohr. For big-data scientists, 'janitor work' is key hurdle to insights. *New York Times*, 2014.

[74] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[76] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, New York, NY, USA, 2016. Association for Computing Machinery.

[77] Lucy Ellen Lwakatare, Aiswarya Raj, Ivica Crnkovic, Jan Bosch, and Helena Holmström Olsson. Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions. *Information and Software Technology*, 127:106368, 2020.

[78] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[79] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[80] Ansong Ni, Daniel Ramos, Aidan ZH Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Soar: A synthesis approach for data science api refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 112–124. IEEE, 2021.

[81] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 61–71, 2017.

[82] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[83] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. TPOT default classifier configuration. https://github.com/EpistasisLab/tpot/blob/master/tpot/config/classifier.py, 2020. Accessed: 2020-03-05.

[84] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017.

[85] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pages 123–137. Springer International Publishing, 2016.

[86] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[87] Rahul Pandita, Raoul Jetley, Sithu Sudarsan, Timothy Menzies, and Laurie Williams. Tmap: Discovering relevant api methods through text mining of api documentation. *Journal of Software: Evolution and Process*, 29(12):e1845, 2017.

[88] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[89] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[90] Valerio Perrone, Huibin Shen, Matthias W Seeger, Cedric Archambeau, and Rodolphe Jenatton. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[91] Philipp Probst, Bernd Bischl, and Anne-Laure Boulesteix. Tunability: Importance of hyperparameters of machine learning algorithms. *arXiv preprint arXiv:1802.09596*, 2018.

[92] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, et al. Data science through the looking glass and what we found there. *arXiv preprint arXiv:1912.09536*, 2019.

[93] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. Machine learning in medicine. *New England Journal of Medicine*, 380(14):1347–1358, 2019.

[94] John W Ratcliff and David E Metzener. Pattern-matching-the gestalt approach. *Dr Dobbs Journal*, 13(7):46, 1988.

[95] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[96] Matthias Reif, Faisal Shafait, Markus Goldstein, Thomas Breuel, and Andreas Dengel. Automatic classifier selection for non-experts. *Pattern Analysis and Applications*, 17(1):83–96, 2014.

[97] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[98] V Rodriguez-Galiano, M Sanchez-Castillo, M Chica-Olmo, and MJOGR Chica-Rivas. Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geology Reviews*, 71:804–818, 2015.

[99] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE, 2017.

[100] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018.

[101] Roni Rosenfeld. A maximum entropy approach to adaptive statistical language modeling. 1996.

[102] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[103] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-FranÃğois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 2015.

[104] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[105] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1171–1188, 2019.

[106] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*, pages 10824–10834, 2019.

[107] Micah J Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. The machine learning bazaar: Harnessing the ml ecosystem for effective system development. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 785–800, 2020.

[108] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959. Curran Associates Inc., 2012.

[109] Xudong Sun, Jiali Lin, and Bernd Bischl. Reinbo: Machine learning pipeline search and configuration with bayesian optimization embedded reinforcement learning. *arXiv preprint arXiv:1904.05381*, 2019.

[110] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[111] Dan Toomey. *Jupyter for data science: Exploratory analysis, statistical modeling, machine learning, and data visualization with Jupyter*. Packt Publishing Ltd, 2017.

[112] TPOT. Github repository, 2018.

[113] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research*, 12(Jul):2411–2414, 2011.

[114] Stef Van Buuren. *Flexible imputation of missing data*. CRC press, 2018.

[115] CJ van Rijsbergen. Information retrieval, 2nd edbutterworths, 1979.

[116] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[117] Manasi Vartak and Samuel Madden. Modeldb: Opportunities and challenges in managing machine learning models. *IEEE Data Eng. Bull.*, 41(4):16–25, 2018.

[118] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. Human-ai collaboration in data science: Exploring data scientists' perceptions of automated ai. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[119] Qianwen Wang, Yao Ming, Zhihua Jin, Qiaomu Shen, Dongyu Liu, Micah J Smith, Kalyan Veeramachaneni, and Huamin Qu. Atmseer: Increasing transparency and controllability in automated machine learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[120] Doris Xin, Eva Yiwei Wu, Doris Jung-Lin Lee, Niloufar Salehi, and Aditya Parameswaran. Whither automl? understanding the role of automation in machine learning workflows. *arXiv preprint arXiv:2101.04834*, 2021.

[121] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. OBOE: collaborative filtering for automl model selection. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 1173–1183. ACM, 2019.

[122] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. Automl pipeline selection: Efficiently navigating the combinatorial space. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[123] Rui Yang, Panos Kalnis, and Anthony KH Tung. Similarity evaluation on tree-structured data. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 754–765, 2005.

[124] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[125] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. Demomatch: Api discovery from demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 64–78, New York, NY, USA, 2017. Association for Computing Machinery.

[126] Yue Zhao, Zain Nasrullah, and Zheng Li. Pyod: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7, 2019.

[127] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 307–318, USA, 2009. IEEE Computer Society.