Learning Repair Rules for Machine Learning Pipelines from AutoML Search Traces

Anonymous Author(s)

Abstract

We frame the task of improving predictive performance of an existing machine learning pipeline by performing a small modification as an analogue to automated program repair. In this setting, the existence of a similar pipeline with better performance, the modification that delivers that improvement, and the task of automatically generating and applying that modification are the analogues of bug, patch, and automated program repair, respectively. We develop a system, Janus, that learns to extract repair rules from a corpus of pipelines produced as a by-product of an automated machine learning search, an approach conceptually similar to learning patches from code corpora. Our experiments show Janus can improve pipelines more often than a baseline approach, with comparable improvements when both succeed, and resulting pipelines closer to the original input pipelines.

1 Introduction

Machine learning has seen remarkable achievements in a wide range of application domains. Many systems employ machine learning pipelines that are built through a carefully designed composition of operators and tuning of hyperparameters. Building these machine learning pipelines requires expert machine learning knowledge, software development experience, and domain expertise.

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 [1, 11, 23]. 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 [17].

A common situation is for a larger system to make use of an *existing* machine learning pipeline, which could be further improved to deliver higher predictive performance. While improving the 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 and thus allowing transparent replacement.

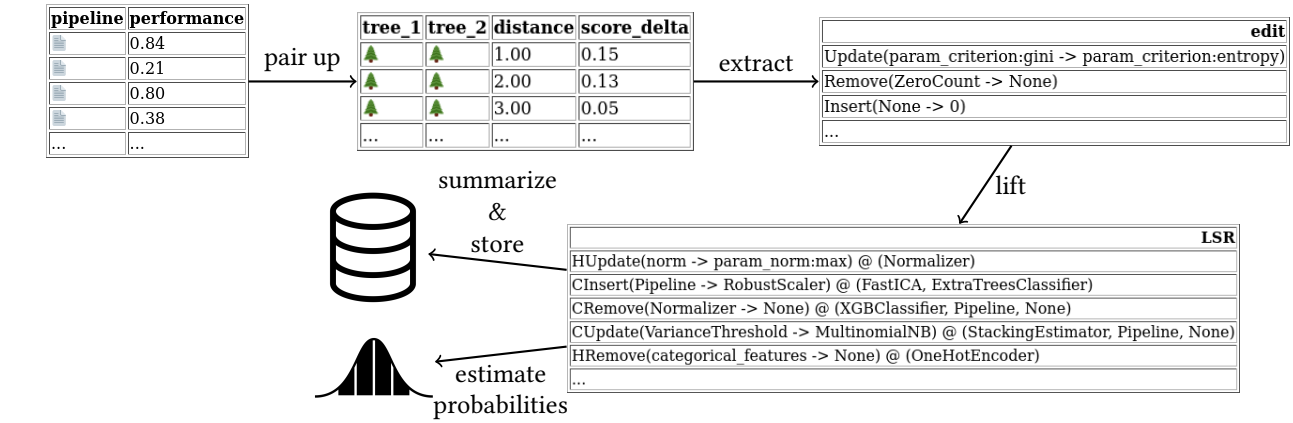
A key insight behind this work is that the scenario described closely aligns with traditional automated program repair [2, 9, 12–16, 18]. In particular, the fact that there may exist a small (as of yet unimplemented) modification to the existing pipeline that would improve predictive performance can 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 the developer to perform repeated and tedious modifications (e.g. tuning hyperparameters, replacing pipeline components) and their evaluation [26], is then a natural analogue to *automated program repair*.

We propose Janus, an automated repair system that transforms machine learning pipelines to automatically improve their predictive performance. To learn these transformations, Janus exploits the large number of pipelines produced as a by-product of existing automated machine learning (AutoML) search procedures. A key insight in Janus is that if we treat pipelines as trees, we can extract candidate transformations as tree edit operations. To this end, Janus introduces the concept of *dominating tree pairs*, where similar pipelines have a performance differential. When extracting pairs, Janus can efficiently prune candidates down by using an approximation of the tree edit distance as a filter. Edit operations extracted from the final set of tree pairs can then be lifted to an abstraction we term *local structural rules*, a typed version of edit operations with ML pipeline specific semantics. Janus summarizes the rules observed into a rule corpus, over which it can compute the joint probability of a given rule and the node at which it is applied. Given a new pipeline, Janus applies transformations in a ranked order to produce candidate repairs. This approach is conceptually similar to existing learned program repair techniques [2, 13, 14].

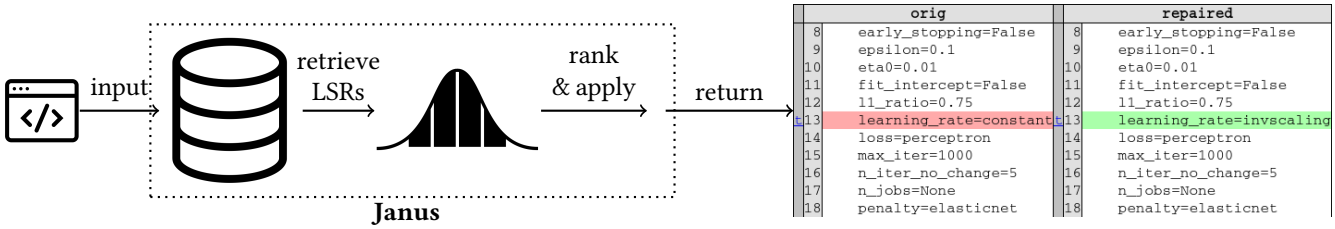
To evaluate Janus, we generate a pipeline corpus using TPOT [20], an AutoML tool that uses genetic programming, on 9 datasets. We compare Janus' ability to improve pipelines to a baseline repair strategy that relies on random mutations. Our results show that 23% – 49% of Janus' repairs result in improved predictive performance, 10 – 27 percentage points more than the baseline, in all 9 datasets. When both approaches improve the predictive performance of the same pipeline, we find that the score improvement is comparable.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) In an offline phase, Janus mines transformations from ML pipelines by identifying tree pairs that may yield improvements, extracting concrete edit operations, and lifting these 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, a single change to the value of the categorical `learning_rate` hyperparameter improved the pipeline's F1-macro score from 0.65 to 0.71.

Figure 1. An overview of Janus, our system to learn repair rules for machine learning pipelines from AutoML search traces.

However, Janus is more likely to produce a repair that is closer in tree edit distance to the original pipeline: only 0.3% of Janus repairs have a tree edit distance greater than 10, compared to 12.7% for our baseline. We also evaluate our system design. We show that when extracting tree pairs, Janus' approximate distance metric outperforms a baseline approach and is comparable to an exact approach. We also show that Janus can effectively summarize repair rules, reducing an original set of transformations by 93.2%. Finally, we carry out a sensitivity experiment, where we replace the initial corpus of pipelines used to learn rules with randomly sampled pipelines. In this setting, we show that Janus is comparable to random mutation, showing that Janus is more likely to outperform when it derives repair rules from a corpus of pipelines produced by a guided rather than random search procedure.

1.1 Janus Overview

Figure 1 presents an overview of our techniques, encompassed in the Janus system. Janus employs two phases. In an offline phase (Figure 1a), the system learns repair rules, which are then applied in an online phase (Figure 1b) to produce repairs.

Building a Pipeline Corpus. The process of learning repair rules starts by collecting a large number of machine learning pipelines. To address the challenge of collecting an appropriately sized corpus, Janus capitalizes on the insight that AutoML search procedures already generate and evaluate many pipelines prior to returning a final pipeline to the user. The collected pipelines and their corresponding performance scores constitute a search trace, which Janus can mine for information. To learn effective repair rules, we identify pipelines that are related 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 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). LSRs 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 the concept

of an *LSR key* (Section 5), and summarizes them by keeping a single rule per LSR key. While summarizing, Janus computes two probability distributions: the conditional probability of applying a transformation with a particular rule key given a tree node, and the marginal probability of transforming a given tree node. 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 enumerate transformed trees in a bounded-depth greedy approach. Janus's repair loop then returns the first transformed and validated tree as a repair.

1.2 Contributions

In this paper we make the following contributions

- *Janus algorithm.* We describe procedures for identifying pipeline pairs for rule learning, extracting and summarizing transformations in the form of rules, and generating transformed pipelines as repairs. In doing so, we present an approximation for tree edit distance which serves to efficiently identify pipeline pairs. We introduce an abstraction, local structural rules, to represent typed edit operations with machine learning pipeline specific semantics. And we show how we use the joint probability over rules and the nodes they transform to rank possible transformations, which then generate repairs.
- *Janus implementation and evaluation.* We implement [1] a version of Janus that repairs pipelines implemented using Scikit-Learn [21], a popular Python machine learning library, and conduct an extensive evaluation. Our results show that repairs generated by Janus are more likely to improve predictive performance than a baseline method. These improvements are comparable in magnitude, but the pipelines produced are more likely to be close to the input pipeline in terms of tree edit distance.

The remainder of the paper is structured as follows. We provide necessary background in Section 2. The core of Janus is described from Section 3 to Section 5. We describe our evaluation methodology in Section 6, and our results in Section 7. Section 8 discusses related work, and Section 9 concludes.

[1] https://anonymous.4open.science/r/67e55703-978b-4977-9b3e-3b9ab9d4f8e8/

2 Background

We introduce machine learning pipelines, automated machine learning, and automated machine learning search traces as core concepts underlying Janus.

2.1 Machine Learning Pipelines

We first briefly describe the use of machine learning pipelines in classification.[2] We assume a setting where the user provides a tabular dataset, $X \in \mathbb{R}^{n \times m}$, consisting of n rows (i.e. observations) and m columns (i.e. covariates). The dataset also contains a vector of $y \in \mathbb{N}^{n \times 1}$ of discrete labels, one for each row in X. The goal of a machine learning pipeline is to learn a function $f : \mathbb{R}^{n \times m} \rightarrow \mathbb{N}^{n \times 1}$ that can *predict* the labels of an input set of observations such that it maximizes predictive performance, measured by an evaluation function $e : \mathbb{N}^{n \times 1} \times \mathbb{N}^{n \times 1} \rightarrow \mathbb{R}$ that computes the quality of the predictions. Pipelines are typically implemented by composing operators from a domain-specific machine learning library and setting appropriate hyperparameters for each operator [21]. As such, the pipeline's function f is drawn from $\mathcal{F}$, the set of pipelines that can be defined by all possible operator compositions and hyperparameter configurations.

2.2 Search Traces in Automated Machine Learning

In many cases, a machine learning developer manually specifies the operators and hyperparameters of a pipeline. However, search-based techniques can be used to more efficiently sample from this exponentially large space of pipelines. These search techniques are broadly termed *automated machine learning* (AutoML). Let (X, y) be a dataset which is further split into training and test portions, $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$. Let b be a compute time budget. Let $c : \mathcal{F} \rightarrow \mathbb{R}$ be a function that computes the time to train/evaluate a pipeline. We abuse notation slightly, such that $f(X, y)$ represents training (i.e fitting) and produces a pipeline f_{fit}, which can then be applied to new data instances ($f_{\text{fit}}(X)$) to produce predictions.

An AutoML system searches within the space of pipelines and optimizes the following:

$$\underset{f \in \mathcal{F}}{\text{argmax}}\ e(f(X_{\text{train}}, y_{\text{train}})(X_{\text{test}}), y_{\text{test}})$$
$$s.t. \sum_{f \in \mathcal{P}} c(f) \leq b,$$

where $\mathcal{P} \subset \mathcal{F}$ is the subset of pipelines evaluated by the system. The subset covered by $\mathcal{P}$ is determined by the search procedure's search space definition and exploration strategy.

An AutoML search procedure will typically produce a series of candidate pipelines, evaluate them, compare their performance to previous pipelines produced, and accumulate the best performing as a final output. We refer to candidates generated and scored as the search trace. Formally,

[2] A similar setup can be used for other supervised learning settings.

Definition 2.1. *AutoML search trace.*

Let S be an AutoML search procedure, $(X_{\text{train}}, y_{\text{train}})$ be a training dataset, $(X_{\text{test}}, y_{\text{test}})$ be an evaluation dataset, e a performance evaluation function, and $\mathcal{P} \subset \mathcal{F}$ be the set of candidate pipelines produced by S during an execution of the search procedure. We define the *AutoML search trace* of S on $(X_{\text{train}}, y_{\text{train}})$ to be the set $\{(f, e(f_{\text{fit}}(X_{\text{test}}), y_{\text{test}}))|f \in \mathcal{P}\}$.

A key insight behind Janus is that we can use this search trace to identify pipeline improvements, in the form of transformation rules. In the following sections we detail how to do so effectively.

3 Building a Pipeline Corpus

We now introduce the process used by Janus to build a corpus of pipelines for rule learning.

3.1 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 $E : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{B}$ 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 represent trees.

Nodes in a tree can be of two types: component nodes ($C \in \mathcal{V}$) or hyperparameter nodes ($\mathcal{H} \in \mathcal{V}$). 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, $\textsc{label} : \mathcal{V} \rightarrow$ 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 $\textsc{parent}$, $\textsc{left}$ and $\textsc{right}$ sibling functions, of type $\mathcal{V} \rightarrow (\mathcal{V} \cup \emptyset)$, and children function $\textsc{children} : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{V})$, produce the expected set of nodes based on tree traversals. Hyperparameter nodes also define a function $\textsc{Value} : \mathcal{H} \rightarrow (\mathbb{R} \cup \mathbb{B} \cup \text{string})$ 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. $\textsc{ToTree} : \mathcal{F} \rightarrow \mathcal{T}$ maps a pipeline to its tree representation, with inverse $\textsc{FromTree} : \mathcal{T} \rightarrow \mathcal{F}$.

Janus defines the distance between two pipelines as the tree edit distance [5] between their respective tree representations, $\textsc{Dist} : \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R}$. When comparing nodes in the tree edit distance computation, Janus uses binary distance over the node labels.

3.2 Corpus of Dominating Tree Pairs

Janus is designed to learn repair rules that can be applied to a pipeline to improve its performance. To extract these rules, Janus requires a corpus of appropriate example pipelines. A key insight is that the appropriate pairs of pipelines are those where they are *nearby* in terms of their structure/values, but one of the pipelines obtains a better predictive performance. To formalize this, we use the concept of pipeline dominance.

For brevity, let $f(D) = f(X_{\text{train}}, y_{\text{train}})(X_{\text{test}})$ mean we fit (i.e. train) a pipeline f on the training data X_{train} and y_{train} and make predictions on the test observations X_{test}.

Definition 3.1. *Pipeline dominance.*

We say a pipeline f_2 *dominates* a pipeline f_1, given a distance constraint d, if $\textsc{Dist}(f_1, f_2) \leq d$ and $e(f_2(D), y_{\text{test}}) > e(f_1(D), y_{\text{test}})$. Intuitively, f_2 is close to f_1 and has improved predictive performance. We refer to the pair $(\textsc{ToTree}(f_1), \textsc{ToTree}(f_2))$ as a dominating tree pair. We refer to f_1 as the pre-pipeline and $\textsc{ToTree}(f_1)$ as the pre-tree. Similarly, we refer to f_2 as the post-pipeline and $\textsc{ToTree}(f_2)$ as the post-tree.

Algorithm 1 illustrates our approach to constructing a corpus of dominating tree pairs. Janus takes an AutoML search trace and samples N_{query} pipelines uniformly at random. We refer to these pipelines as *query pipelines*. The goal is to pair each query pipeline with a set of at most k pipelines that dominate it.

Computing the edit distance for all pairs of pipelines in the trace is expensive, with the distance operation having time complexity of $O(n^2 m(1 + log\frac{m}{n}))$ given a tree with n nodes and another with m nodes [6]. To address this challenge, Janus exploits the fact that a key property of dominance is a better predictive performance, and first filters dominating candidates down to those that have higher performance score. Janus then further refines this set of candidates to the top N_{maybe} pipelines that are *possibly* dominating based on an approximate distance metric.

To compute our approximate distance metric (Algorithm 2), 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. Our approximate distance is defined as the Euclidean distance between two such count vectors. Janus takes the N_{post} candidates with the smallest such distance to the query pipeline under consideration.

Finally, Janus computes the exact tree-edit distance on this smaller set of candidate pipelines, and keeps any pipeline that

has a distance less than or equal to our defined threshold d. Each of these post-pipelines is paired with the pre-pipeline to produce a dominating tree pair. Janus repeats this process for the N_{query} pipelines initially sampled.

Algorithm 1 Sampling dominating tree pairs for a single query pipeline

INPUT: A search trace S; 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 *possibly* dominating pipelines to sample; an integer d for the maximum tree edit distance for a dominating tree; and an integer k for the maximum number of dominating tree pairs to produce per query pipeline.
OUTPUT: At most k dominating tree pairs
procedure SAMPLETREEPAIRS
▷ *Dominating trees must have better score*
$candidates \leftarrow \{t \mid (t,s) \in S \setminus (t_q, s_q) \wedge s > s_q\}$
▷ *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})
▷ *Satisfy distance threshold, so are dominating*
$dominating \leftarrow \{t \mid \text{DIST}(t,t_q) \leq d\}$
$dominating \leftarrow$ TAKE($dominating$, k)
return $\{(t_q,t) \mid t \in dominating\}$

Algorithm 2 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).

4 From Edits to Local Structural Rules

In Janus, we propose that we can repair a pipeline by learning from the operations required to transform the pre-pipeline to the post-pipeline in a dominating tree pair. These basic operations are defined to be the sequence of update, insert, and delete operations computed for purposes of the tree edit distance [5]. 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 4.1. *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 element is the pre-tree, rooted at the target location of the transformation, and the third element 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 dominating 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 : 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 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,

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

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

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

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

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

Figure 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.

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 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 rules to learn to rank them, given an input tree node. This ranking is used by a lazy tree generator, which produces candidate transformed trees consumed by the core repair algorithm. We now provide details of this process.

5.1 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 dominating 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.

To do so we introduce the concept of an LSR key.

Definition 5.1. *LSR key.*

An LSR key is a function $\textsc{Key} : \text{LSR} \rightarrow \mathcal{L} \times \text{string} \times \mathcal{P}(\text{string}) \times \text{string}$, 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.

Given a collection of LSRs, Janus greedily summarizes the collection by keeping the LSR with the largest performance change for each unique LSR key. The performance change is computed as the difference in performance score between the pipelines associated with the dominating tree pairs. This heuristic use of performance change, which can be extracted directly from the search trace, 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 (denoted as $P(\text{rule-key}|\text{node})$) and the marginal probability of observing a given pre-tree node over all rules (denoted as $P(\text{node})$). Both of these probabilities can be computed by simply counting and normalizing appropriately; we elide their definitions here for brevity.

5.2 Ranking and Applying Rules

Janus collects a map from pre-tree label to the set of LSRs with that corresponding pre-tree label[3] 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[4] is summarized in Algorithm 3.

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

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

Algorithm 3 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 $P(\text{node})$; a conditional probability function CondProb that computes $P(\text{rule-key}|\text{node})$; 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) | r ∈ R(n), n ∈ N}
    ▷ Remove LSRs that can't be applied based on pre-conditions
    candidates ← {(r,n)| n,r ∈ candidates ∧ CanApply(r,n)}
    ▷ Sort with joint probability function
    ranked ← SortBy(candidates, λ(r,n) : CondProb(Key(r),n) * MarginalProb(n))
    return ranked
```

Janus, given an input tree, produces a (lazy) tree generator which a downstream repair step can query for new candidate transformed trees. This tree generator works on a tree queue and proceeds in a step-wise procedure. At each step, the generator takes the first tree in the queue and yields it to the caller. After yielding a tree, the generator derives k new trees by applying ranked transforms. These k trees are enqueued for future steps. When deriving transformed trees, the tree generator excludes any rules used to derive the current version of the tree. The goal of this exclusion is to avoid repeated applications of the same transform. This procedure is summarized in Algorithm 4.

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, tests whether it will produce a runtime exception by running it on a sample of the user's dataset, and if no exception is raised it returns the associated pipeline. This broadly adheres to the *generate-and-validate* approach, which has been shown to be effective in traditional automated program repair [13, 14, 16]. To limit the time spent in the repair loop, Janus takes a time budget (set to 60 seconds by default). If no repair validates during this time, Janus returns a null pipeline. Algorithm 5 summarizes this repair procedure.

5.3 From Scripts to Pipelines

In practice, machine learning pipelines are often written as part of larger ad-hoc experimental scripts or computational notebooks [25]. These artifacts will typically perform additional steps, beyond just building/training 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

Algorithm 4 Janus lazy tree enumerator.

INPUT: An input tree t; Janus's CanCompile which tries to lower a tree to its pipeline representations (using FromTree) and returns true if it succeeds without any pipeline building exceptions; an integer bound k on the number of trees to generate per step in the tree generator; 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 transform rule to it.
OUTPUT: A lazy generator for transformed trees.

```
procedure TreeGenerator
    ▷ Queue of derived trees, starts with input
    q ← Queue(t)
    ▷ Track transforms used to derive, avoid reapplying
    PastRules ← Map()
    ▷ Input tree has no prior transforms
    PastRules[q] ← ∅
    while q ≠ ∅ do
        h ← PopHead(q)
        ▷ Limited set of transforms excluding prior ones
        transforms ← RankTreeTransformations(h)
        transforms ← {(r, n)|(r, n) ∈ transforms, r ∉ PastRules[h]}
        transforms ← Take(transforms, k)
        for (r,n) ∈ transforms do
            h' ← Apply(h,n,r)
            ▷ Janus checks for pipeline compilation validity
            if CanCompile(h) then
                yield h
                Enqueue(q,h)
    ▷ null tree as sentinel
    return ∅
```

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 tracer. Our front-end first converts notebooks to scripts. We then extract a source-line-level dependency graph based on executing the program and tracking the memory address of definitions and uses.[5] 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

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

Algorithm 5 Janus high-level repair procedure.

INPUT: An ML 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 Fit which attempts to fit the pipeline to a dataset; an integer bound k on the number of trees to generate per step in the tree generator; a sample of the user's dataset (X,y); a current time function TimeNow, and a time budget 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,k)
    timeLeft ← b
    ▷ Limit repair time for responsiveness
    while timeLeft > 0 do
        start ← TimeNow()
        ▷ Next tree in the generator's queue
        t' ← gen.next()
        ▷ null pipeline if no more transforms possible
        if t' == ∅ then
            return ∅
        ▷ Lower to pipeline
        p' ← FromTree(t')
        try
            ▷ Validate if raises exceptions on sample data
            Fit(p',X,y)
            return p'
        catch Exception
            ▷ If exception, just account for time and continue
            spent = TimeNow() − start
            timeLeft −= spent
            continue
        end try
    ▷ null pipeline if no repair produced
    return ∅
```

constructed based on the script contents. These pipelines are then given to Janus's repair module, as detailed previously.

6 Experimental setup

We thoroughly evaluate Janus on several dimensions, focusing on its utility for repairing pipelines. We first describe our experimental setup.

6.1 Pipeline corpus

For our evaluation, we use the nine datasets in the TPOT evaluation corpus [19], and we produce a pipeline corpus by running TPOT [20], a genetic programming AutoML tool, to obtain search traces.

For each dataset, we use 50% of the data as a development set, further splitting 80% for training and remainder for validation. The other 50% of the data is held-out to be used as a test set for evaluating Janus' performance compared to a baseline method in Section 6.4.

For each dataset, we run TPOT for one hour using its default configuration. For each pipeline considered by TPOT during its search, we compute its score on the validation set using macro-averaged F1. We obtain a total of 11,134 total pipelines paired with their scores on the validation set.

6.2 Extracting tree pairs

From this pipeline corpus, we extract dominating tree pairs (Section 3.2). For each dataset, we sample scored pipelines and convert them to the corresponding tree representation, 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 17,023 dominating tree pairs.[6]

6.3 Extracting rules

From the dominating tree pairs, we extract rules that will be applied during pipeline repair. First, we filter the pre/post-tree pairs to those that have at most an edit distance of $d = 10$, a distance that is large enough to allow component changes, but small enough so that edits do not require many composite operations. From this subset, we extract rules and summarize them to compute a rule corpus following the approach described in Section 4 and Section 5. This results in 38,023 raw LSRs, which Janus summarizes to produce a rule corpus of 2,581 rules.

6.4 Random mutations baseline

We evaluate pipeline repairs learned by Janus against a baseline of a pipeline repair that uses random mutations. The *RandomMutations* method works by instantiating Janus's RankTreeTransformations (see Algorithm 3) with a random ranking of transformations over all nodes in the tree, with the LSR for each node instantiated by randomly choosing from Janus's LSR types (based on the node type) and drawing a post-tree value from TPOT's search space for that corresponding component or hyperparameter. This sampling procedure is repeated each time a new repair attempt takes place.

From the evaluation pipeline corpus (Section 6.1), we sample 100 pipelines for each dataset as our evaluation targets. We hold one dataset as fixed and then evaluate Janus' and *RandomMutations*' pipeline repairs on this dataset. For Janus, we use only rules produced for all *other* datasets, excluding the one under consideration, in order to evaluate whether learned rules generalize to new situations. Both Janus and *RandomMutations* reserve 5% of the development set to test for runtime errors, and generate candidate pipelines until the first pipeline to raise no errors is obtained, or a 60 second budget is exceeded. Once a pipeline repair is yielded, we evaluate it on the held-out 50% test set by computing the macro-averaged

[6] A a pipeline may be dominated by fewer than 10 other pipelines, thus the total tree pairs is less than 18,000.

F1-score using 5-fold cross-validation. If a method does not produce any acceptable pipeline, we record a score of nan.

7 Results

We present our evaluation results, first focusing on broader system design choices in Janus, followed by repair performance, sensitivity of Janus to the original pipeline corpus, and distance of repairs.

7.1 System Design

We evaluate the impact of our approximate distance metric (Algorithm 2) on the distance distribution for dominating tree pairs. In particular, we compare the use of our approximate distance metric in tree sampling to a uniform random sampling approach and an extract distance metric approach. For the exact distance 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 3 shows the empirical cumulative distribution function (ECDF) for the distance of dominating tree pairs produced using all three methods. Our results show that our approximate distance approach strictly improves on random sampling, and very closely matches the results obtained with the exact approach. Additionally, we found that our approximate approach had an average runtime of 759.26 ($\sigma = 282.87$) seconds per dataset, compared to an average 9,967.94 ($\sigma = 7{,}759.96$) seconds for the exact method and an average 813.41 ($\sigma = 276.23$) seconds[7] for the random sampling method.

To produce a rule corpus, Janus first lifts edit operations to local structural rules (LSRs). In this process, Janus mines a total of 38,023 LSRs. To effectively generate tree transforms, Janus summarizes this set of LSRs to final corpus to 2,581 rules, relying on the LSR keys (Definition 5.1) and a greedy score heuristic. This summarization reduces the initial set of LSRs by 93.2% and normalizes the initially skewed distribution of LSR types from a large number of HUpdate rules to a more balanced mix. Figure 4 illustrates the distribution of LSR types before and after summarization.

7.2 Performance

Next we evaluate Janus' ability to provide useful pipeline repairs. Figure 5 shows the fraction of pipelines produced by Janus and *RandomMutations* that improve the original pipeline's score. We find that Janus consistently outperforms *RandomMutations*, improving up to 49% of pipelines and improving upon the baseline by 10 – 27 percentage points.

[7] 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.

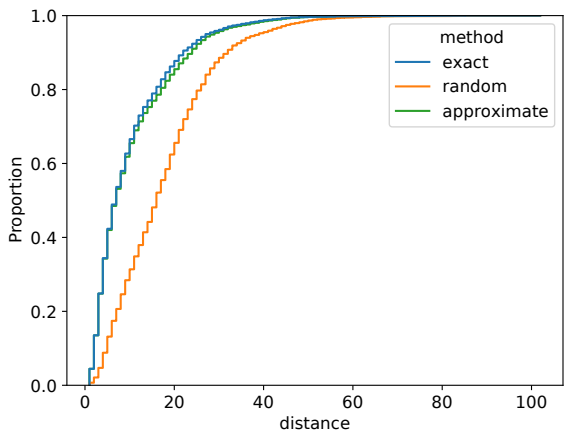
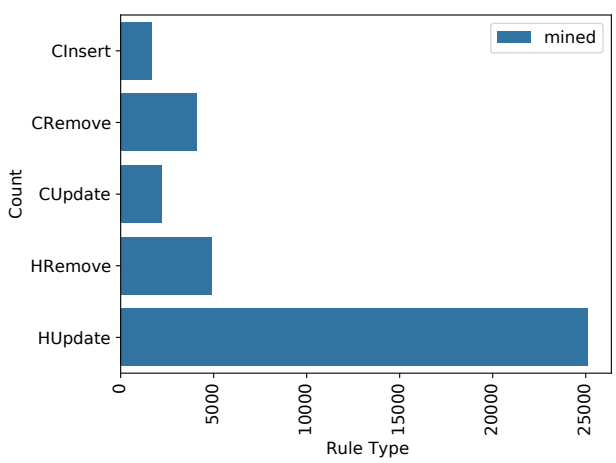


Figure 3. Janus' approximate distance metric is used to efficiently identify dominating tree pair candidates. We show that this approximation yields better results than random sampling and closely tracks an exact distance method, while being faster than both the exact and random methods.

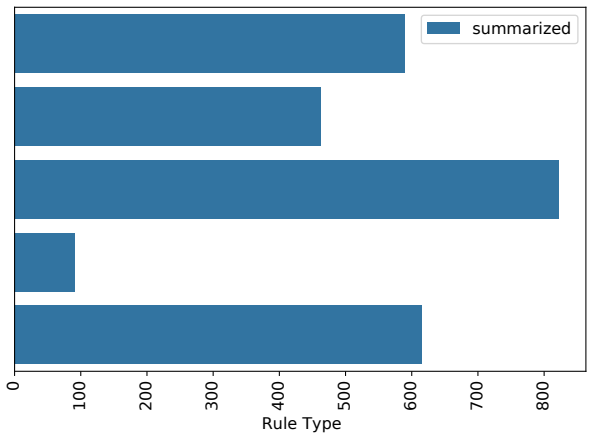
Though Janus performs well by this measure, there is still a substantial fraction of pipelines for which performance does not improve. In Figure 6 we investigate this further by plotting the ECDF of the improvement in score between the repaired pipeline and the original pipeline. For purposes of computing the change in score, if a repaired pipeline fails to execute or no repair was produced, the repaired pipeline is assigned a score of 0.0, and the resulting score change is the negative of the original pipeline's score. We find that Janus produces a sizable fraction of repaired pipelines that have no impact on pipeline score ($\Delta \approx 0$), but that it rarely produces "repairs" that actually hurt performance. In contrast, the distribution of score changes produced by *RandomMutations* is more evenly centered around 0, as any given mutation might be just as likely to help or hurt pipeline performance, and it produces a substantial fraction of repairs that result in lower performance.

Next, we restrict our evaluation to the cases where both Janus and *RandomMutations* improve performance for the same original pipeline, a closer look at the rightmost part of the distributions in Figure 6. Figure 7 shows the average score improvement and bootstrapped 95% confidence interval for each approach. Our results show that the improvements produced by both methods on this subset of pipelines are comparable.

We formalize these comparisons in Table 1. We test the null hypotheses that Janus and *RandomMutations* produce the same number of pipeline improvements (i.e. successful repairs) or the same magnitude of score changes when there is an improvement. For the first test, we use the McNemar paired test which is a non-parametric test that handles the fact that both systems are producing repairs for the same set of original pipelines, and thus we cannot assume independence as is standard in other statistical tests. For the second test, we use



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



(b) Distribution of types for LSRs after Janus has summarized them using a key-based approach.

Figure 4. Janus mines a total of 38,023 local structural rules (4a), which are then summarized to a rule corpus of 2,581 local structural rules using the LSR keys (4b). Janus's summarized rule corpus presents a more balanced mix of rule types.

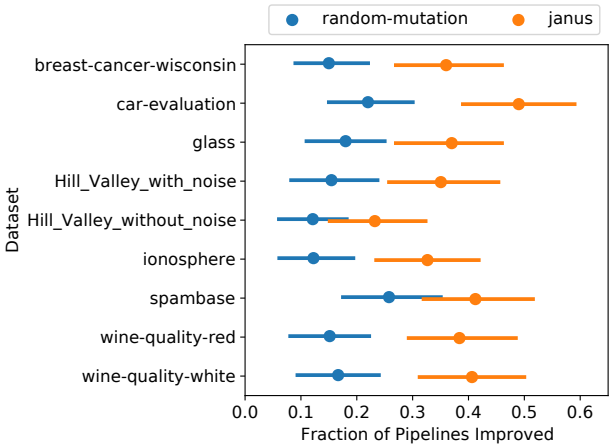


Figure 5. Janus repairs improve on the original pipeline (i.e. increase predictive performance) more often than the repairs produced by *RandomMutations*. We show the fraction of improved pipelines, with bootstrapped 95% confidence intervals for both approaches.

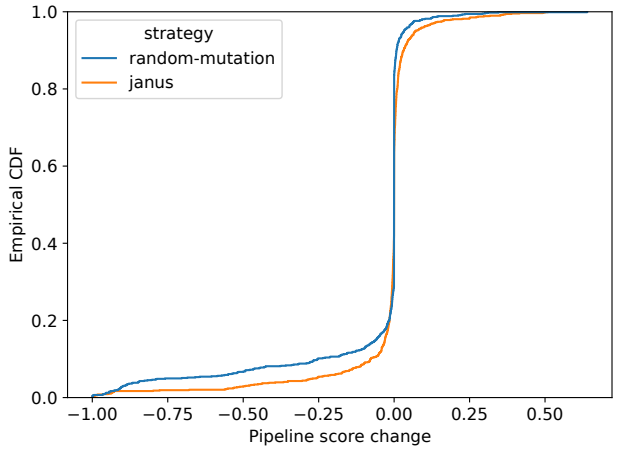


Figure 6. ECDF of the score change between the original pipeline and the repaired pipeline for both approaches. Janus is less likely to produce a repair that actually reduces performance, as compared to *RandomMutations*.

a paired t-test. In all cases we perform Bonferroni corrections for multiple comparisons. We reject the null hypothesis of an equal number of improvements at the $p < 0.01$ level, but we cannot reject the null hypothesis of a mean difference in score change of zero when both methods produce an improvement.

7.3 Sensitivity to Pipeline Corpus

Learned pipeline repairs may be only as effective as the rules that can be identified in the pipeline corpus. To evaluate the sensitivity of Janus to its pipeline corpus, we conduct an experiment in which we follow the methodology in Section 6

Test	Statistic	P-Value	Evaluating
McNemar Paired Test	77	*** (<0.01)	Number of repairs
Paired t-test	-1.05	ns (>0.05)	Score improvement

Table 1. We carry out a statistical test for the number of pipeline improvements and the score change when an improvement is made. We find that Janus produces more improvements ($p < 0.01$), but we cannot reject the null hypothesis that the mean difference in score changes is zero across methods when both methods produce a successful repair.

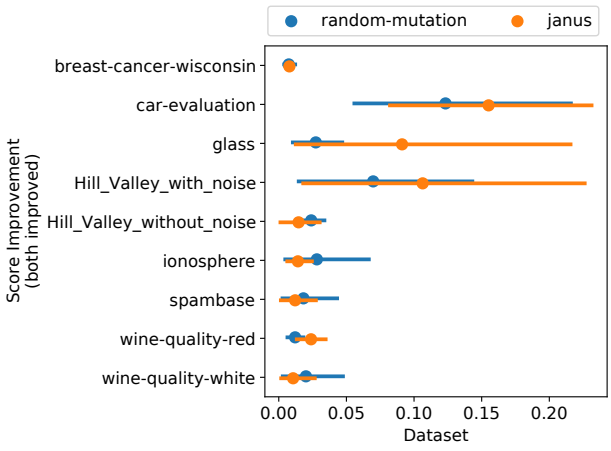
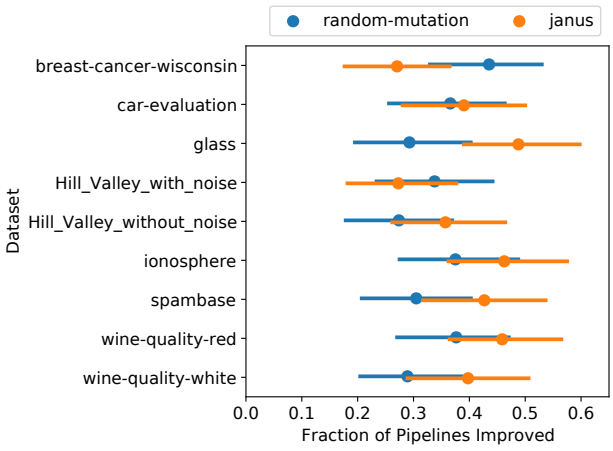
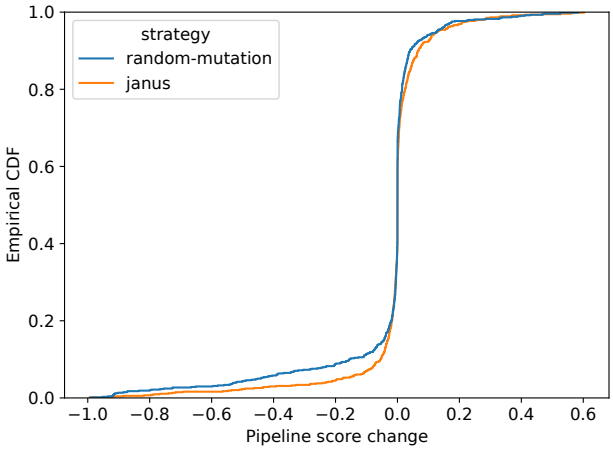


Figure 7. When both Janus and *RandomMutations* improve the performance of the original pipeline, we find that Janus provides a comparable improvement in score.



(a) Fraction of pipelines improved, with bootstrapped 95% confidence intervals



(b) ECDF of score changes

Figure 8. When using an alternative random search to generate the pipeline corpus over which Janus learns rules, we find that Janus, while still able to improve 27%–48% of pipelines per dataset, does not provide the same improvement over *RandomMutations* as when the corpus consists of pipelines produced through genetic programming.

to create the pipeline corpus but instead of using TPOT to generate pipelines, we use a random pipeline search process. This search strategy is customized for joint pipeline structure and hyperparameter configuration search. 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 search space (`tpot.config.classifier_config_dict`). [8] The rest of the setup is identical. Using this alternative pipeline corpus, we evaluate 10,498 pipelines and scores, extract 36,254 LSRs, and compile a summarized rule corpus of 4,613 rules.

In Figure 8, we find that Janus still produces improvements in 27%–48% of repairs per dataset. However, in this setting we find that we cannot reject the null hypothesis that both Janus and *RandomMutations* provide the same number of pipeline improvements, nor that the mean difference in score change (when there is an improvement) is 0.0. This may indicate that Janus is more effective at extracting rules when the pipeline corpus is generated by a guided (rather than random) process.

7.4 Repair distance

As both Janus and *RandomMutations* are shown to produce repairs that improve pipeline performance, we would like to better understand these repairs. Figure 9 shows the ECDF of tree edit distance (with respect to the corresponding original pipeline) for pipelines improved by the corresponding system. Practically all (99.7%) of Janus' repairs have distance less than or equal to 10 as expected, given that this distance constraint is enforced in the dominating tree pair extraction procedure (Section 6.2). However repairs produced by *RandomMutations* have a much longer tail of edit distance with 12.7% having a distance greater than 10. These large edit repairs represent significant modifications to the original pipelines and might be conceived of as random samples from the space of pipelines. Janus is built on the view that repairs should both improve predictive performance and have a small edit distance.

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

8 Related Work

We discuss related work in automated machine learning, automated program repair, and rule mining in the context of programming languages and software engineering.

AutoML. We focus our discussion of AutoML to tools that generate classical ML pipelines. Recent years have seen an

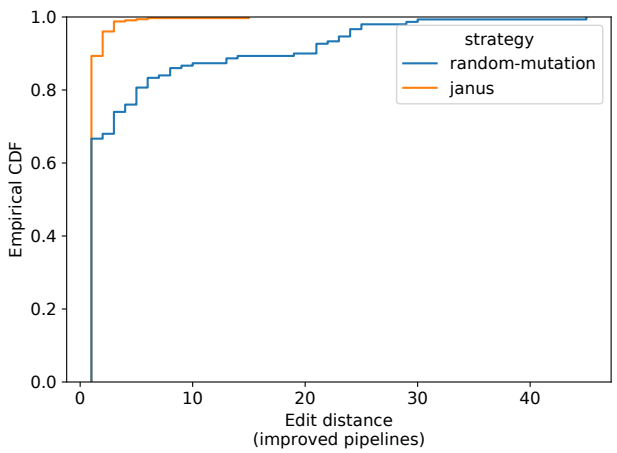


Figure 9. ECDF of tree edit distance for improved pipelines shows that Janus' repairs are consistently closer to the original pipeline compared to those produced by *Random-Mutations*. Only 0.3% of Janus' repairs have a distance greater than 10, compared to 12.7% of repairs by *RandomMutations*.

increase in the number of systems and search procedures used to tackle the task of pipeline generation. TPOT [19] uses genetic programming to generate tree-based pipelines. Auto-Sklearn [8], a model-based algorithm configuration system, combines Bayesian optimization with meta-learning to produce ensembles of automatically configured ML pipelines. AutoBazaar [24] provides an AutoML system, based on multi-armed bandits and Bayesian optimization, built on top of a novel library of composable ML primitives and curated pipeline templates. OBOE [27] and TensorOBOE [28] approach AutoML as an active learning and matrix completion problem, where pipelines can be chosen based on the estimated performance given a smaller set of observed pipeline performance results. In the area of programming languages and software engineering, AL [4] uses dynamic program analysis to learn to generate new pipelines, and AMS [3] allows users to automatically generate an AutoML search space based on an initial set of pipeline components. LALE [10] implements a domain specific language to define type-based search spaces for AutoML, and uses these types to guide a search procedure for new pipelines. There has also been an increased interest in neural architecture search methods which generate new neural network architectures [7], but these are outside of the scope of our focus on classical ML pipelines.

Janus, like these systems, produces new machine learning pipelines to automatically improve predictive performance. However, in contrast to existing work, Janus is focused on a different setting: one where there is already an *existing* ML pipeline, which we would like to improve through a small change. This scenario stands in contrast with the traditional AutoML setting, where a user only provides their dataset and then allows a long-running search procedure to generate promising pipelines. To tackle the challenge of small modifications to pipelines Janus frames pipeline improvement as a program repair task and learns transformation rules.

Automated Program Repair and Rule Mining. Automated program repair (APR) techniques aim to reduce (or remove) the amount of time human developers spend on writing software patches by automating the process of developing and testing fixes. In particular, the *generate-and-validate* approach implemented in recent systems has been shown to successfully repair real bugs in large software projects written in production languages such as C and Java [22].

GenProg [9] introduced the use of genetic programming for automated repair of C bugs. Prophet [16] developed a novel ranking of hand-designed templatized patches based on a set of learned weights over patch features and the target application context. Genesis [14] improved over prior work by introducing an procedure for *learning* patches from different code versions rather than relying on hand-designed patch templates. GetAFix [2] learns a hierarchical clustering and ranking of fix templates based on existing human edits. FixMiner [13] learns to extract tree-based edit scripts for purposes of automatically repairing programs.

Janus, similarly to this body of prior work, learns transformations to produce repaired versions of the input pipeline. Janus ranks these transforms by relying on simple joint probabilities, and takes a generate-and-validate approach to producing the output repair, returning the first fix that does not raise runtime exceptions on a sample of the dataset. In contrast to traditional APR, Janus is not trying to repair a discrete software bug, but rather the "bug" in this setting is lower then desired predictive performance and the goal is to improve on this continuous metric. Rather than rely on a test suite to validate a repair, Janus uses a small sample of the data to ensure no runtime errors are produced, but performs no additional validation, which enables fast responses. The framing of AutoML as a program repair task is a key novel perspective.

9 Conclusion

We frame the problem of having an existing machine learning pipeline which could improve in predictive performance through a (as of yet unimplemented) small change, a corresponding modification that achieves the improved performance, and a procedure for automatically generating such a modification as analogues to a bug, patch, and automated program repair. We present Janus, a system that learns repair rules for machine learning pipelines by analyzing a large corpus of pipelines produced as a by-product of an AutoML search procedure. We show that Janus can improve more pipelines than our baseline, the increases in predictive performance are comparable, and the resulting Janus pipelines are more likely to be close in tree edit distance to the input pipelines.

References

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

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[3] José P. Cambronero, Jürgen Cito, and Martin C. Rinard. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 763âĂŞ774. https://doi.org/10.1145/3368089.3409700

[4] José P. Cambronero and Martin C. Rinard. 2019. AL: Autogenerating Supervised Learning Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 175 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360601

[5] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2007. An optimal decomposition algorithm for tree edit distance. In *International Colloquium on Automata, Languages, and Programming*. Springer, 146–157.

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

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21. http://jmlr.org/papers/v20/18-598.html

[8] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[9] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[10] Martin Hirzel, Kiran Kate, Avraham Shinnar, Subhrajit Roy, and Parikshit Ram. 2019. Type-Driven Automated Learning with Lale. *arXiv preprint arXiv:1906.03957* (2019).

[11] F. Ishikawa and N. Yoshioka. 2019. 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)*. 2–9. https://doi.org/10.1109/CESSER-IP.2019.00009

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

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

[14] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727âĂŞ739. https://doi.org/10.1145/3106237.3106253

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

[16] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2837614.2837617

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

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

[19] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (Denver, Colorado, USA) *(GECCO '16)*. ACM, New York, NY, USA, 485–492. https://doi.org/10.1145/2908812.2908918

[20] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. 2016. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*. Springer International Publishing, Chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, 123–137. https://doi.org/10.1007/978-3-319-31204-0_9

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

[22] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 24–36.

[23] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-FranÃğois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems* (2015).

[24] Micah J. Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. 2020. The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 785–800. https://doi.org/10.1145/3318464.3386146

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

[26] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. 2018. Accelerating human-in-the-loop machine learning: Challenges and opportunities. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[27] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. 2019. OBOE: Collaborative Filtering for AutoML Model Selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 1173–1183.

[28] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA.