Mining Nearby Repairs that Improve Machine Learning Pipeline Performance

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 mines repair rules from a large corpus of pipelines, an approach conceptually similar to learning patches from code corpora. Our experiments show Janus can improve performance in 16% – 42% of the test pipelines in our experiments, outperforming baseline approaches in 7 of the 9 datasets in our evaluation.

ACM Reference Format:
Anonymous Author(s). 2021. Mining Nearby Repairs that Improve Machine Learning Pipeline Performance. In *Proceedings of The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, 8, 26]. 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 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 [11].

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 key insight behind this work is that the scenario described closely aligns with traditional automated program repair [2, 7, 10, 12, 14–16, 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 developer intervention is then a natural analogue to *automated program repair*.

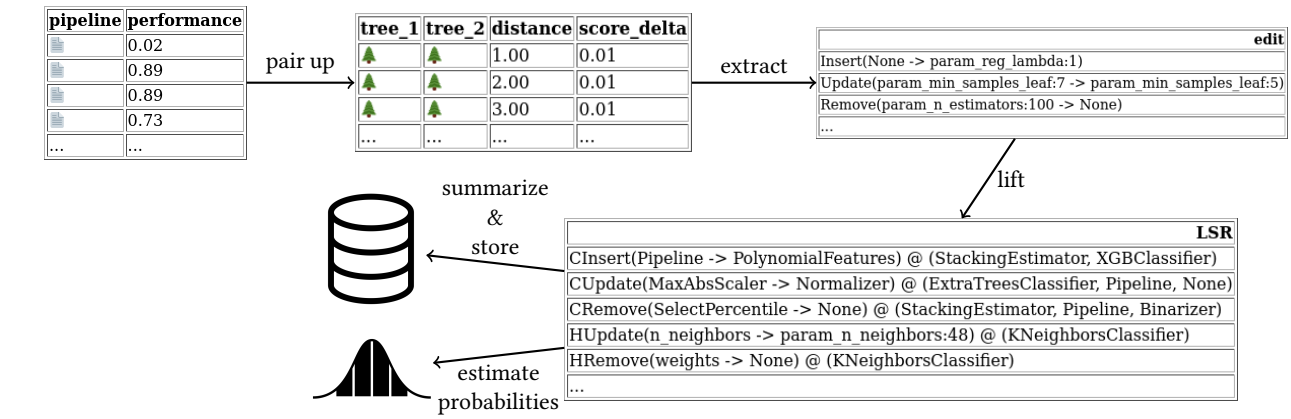
We propose Janus, an automated repair system that mines nearby transformations for machine learning pipelines, which when applied can automatically improve their predictive performance. To mine these transformations, Janus first collects a large number of machine learning pipelines and their scores on 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 [9]. 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 [2, 12, 14].

To evaluate Janus, we collect a corpus of pipelines generated using an off-the-shelf genetic programming AutoML tool, TPOT [20]. 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.

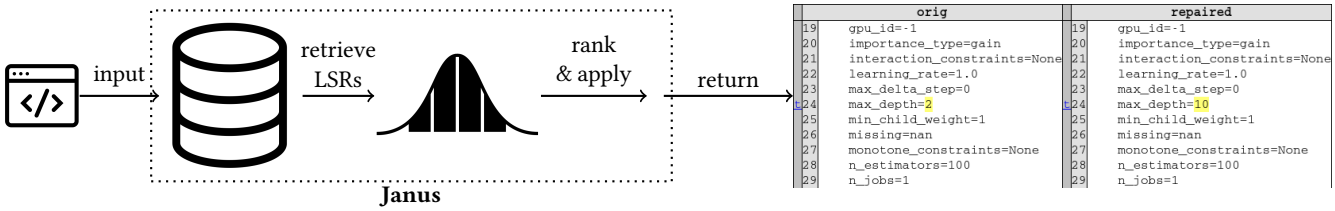
Our results show that Janus can improve 16% – 42% of the pipelines across our test datasets, more than baseline approaches in 7 of our 9 datasets. We also evaluate our system design. We show that when

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece

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 pairs of nearby ML pipelines with a performance differential, extracts concrete edit operations, and liftes 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 1: An overview of Janus, our system to learn single-step repair rules for machine learning pipelines.

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 initial corpus of pipelines used to learn rules with pipelines produced using random search. 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 from which we mine rules.

1.1 Janus Overview

Figure 1 presents an overview of Janus's two phases. In an offline phase (Figure 1a), the system mines repair rules, which are then applied in an online phase (Figure 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 [9] 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 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, 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 greedily enumerate transformed trees. Janus returns as a repair the *first* transformed tree that satisfies the specified distance bound and does not generate any runtime exceptions. In particular, note that Janus *does not* repeatedly generate, fit, and evaluate candidate pipelines.

1.2 Contributions

In this paper we make the following contributions

- *Janus algorithm.* We describe 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 as repairs. 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 to produce 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 three baseline in 7 of our 9 datasets.

The remainder of the paper is structured as follows. The background and core of Janus is described from Section 2 to Section 4. We describe our evaluation methodology in Section 5, our results in Section 6, and threats to validity in Section 7. Section 8 discusses related work, and Section 9 concludes.

2 BUILDING A PIPELINE CORPUS

First, we review the concept of machine learning pipelines within the context of Janus, as well as tree-based representation of pipelines.

2.1 Supervised 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 $f : \mathbb{R}^{n \times m} \to \mathbb{N}^{n \times 1}$ that can *predict* the labels of an input set of observations such that it maximizes predictive performance, measured by an evaluation function $e : \mathbb{N}^{n \times 1} \times \mathbb{N}^{n \times 1} \to \mathbb{R}$ 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 prediction function f is drawn from $\mathcal{F}$, the set of functions that can be defined by all possible operator compositions and hyperparameter configurations.

2.2 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 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, $\textsc{label} : \mathcal{V} \to$ 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. $\textsc{ToTree} : \mathcal{F} \to \mathcal{T}$ maps a pipeline to its tree representation, with inverse $\textsc{FromTree} : \mathcal{T} \to \mathcal{F}$.

Janus defines the distance between two pipelines, $\textsc{Dist} : \mathcal{F} \times \mathcal{F} \to \mathbb{R}$, as the tree edit distance [4] 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 2.1. 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, 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 $\textsc{Dist}(f, f') \leq d \wedge e(f(X,y)(X'),y') < e(f'(X,y)(X'),y')$.

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

2.3 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.[3] While such tools, typically termed AutoML systems, are interesting in their own right, Janus uses their search as a simple way of collecting many pipelines. 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 as mining experiment tracking frameworks like ModelDB [31] or collaborative machine learning websites like OpenML [30].

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

[1] https://anonymous.4open.science/r/67e55703-978b-4977-9b3e-3b9ab9d4f8e8/
[2] A similar setup can be used for other supervised learning settings.
[3] 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.

Algorithm 1 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 $O(n^2 m(1+log\frac{m}{n}))$ given a tree with n nodes and another with m nodes [5]. 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.

To compute an 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. 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 [9], 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 the N_{query} pipelines initially sampled.

Algorithm 1 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 *possibly* 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})
▷ *Satisfy distance threshold, so are d-repairs*
repairs $\leftarrow \{t \mid \text{Dist}(t, t_q) \leq d\}$
repairs $\leftarrow$ Take(*repairs*, k)
return $\{(t_q, t) \mid t \in repairs\}$

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).

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 [4]. 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 3.1. 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 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 2 presents Janus' LSRs in terms of inference rules. In general, an LSR checks that the node is of the appropriate type (i.e. C or $\mathcal{H}$), is not a no-op change (where the node already has the value that

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

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

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

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

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

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.

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.

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 rules 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 the details of this process.

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 the concept of an LSR key.

Definition 4.1. 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.

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. 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 $P(\text{rule-key}|\text{node})$) and the marginal probability of observing a given pre-tree node label 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.

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[4] 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

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

and target application location, each of which constitutes a possible repair. This procedure[5] is summarized in Algorithm 3.

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 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 4.

Algorithm 4 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'
    ▷ null tree as sentinel
    return ∅
```

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 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 5 summarizes this repair procedure.

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

Algorithm 5 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 ∅
        ▷ Repair too far away
        if Dist(t, t') > d then
            continue
        ▷ Lower to pipeline
        f' ← FromTree(t')
        try
            ▷ Check possible exceptions on sample data
            Fit(f', X, y)
            return f'
        catch Exception
            continue
        end try
    ▷ null pipeline if no repair produced
    return ∅
```

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 [28]. 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 tracer. Our front-end first converts 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.[6] Janus's front-end identifies nodes in the graph involving our target ML library

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

(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 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 2.1). 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.

5.1 Pipeline corpus

For our evaluation, we use the nine datasets in the TPOT evaluation corpus [19]. For each dataset, we produce a pipeline corpus by running TPOT [20], 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.

5.2 Extracting tree pairs

From this pipeline corpus, we extract d-repairs (Section 2.3) 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.[7]

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 3 and Section 4. This results in 40,232 raw LSRs lifted from edit operations, which Janus summarizes to produce a rule corpus of 2,939 rules.

5.4 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 2, and uses a random forest regression model to predict the corresponding score. We use the random forest regression implementation available through Scikit-Learn [21] (version 0.22.2) with default hyperparameters.

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).

5.6 Evaluating candidate repairs

For each dataset, we sample 100 pipelines from the pipeline corpus (Section 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.

6 RESULTS

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

6.1 System Design

We evaluate the impact of the approximate distance metric (Algorithm 2) 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

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

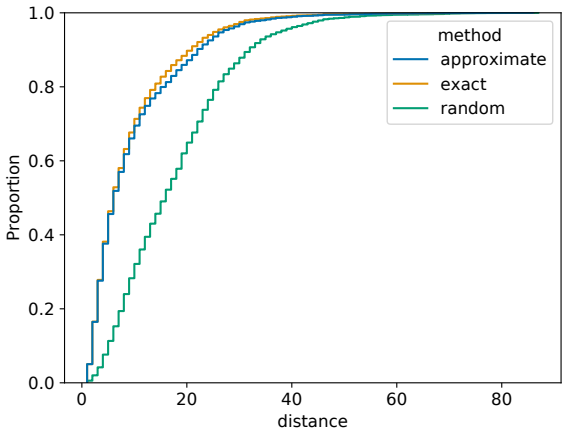


Figure 3: The approximate distance metric used by Janus to efficiently identify d-repair candidates for rule mining. This approximation closely tracks an exact distance method, while being substantially faster. This result aligns with that derived by Jiang et al [9] and Yang et al [34],

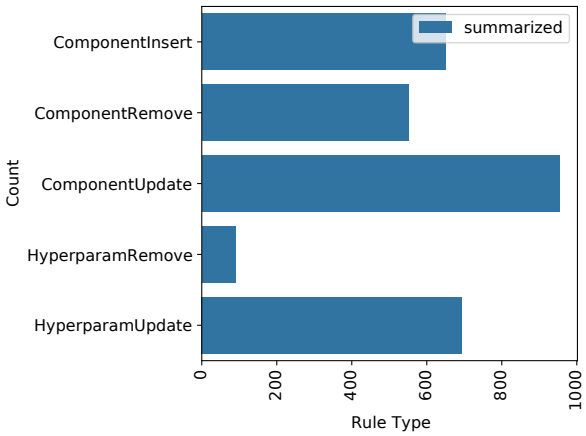


Figure 4: Janus mines a total of 40,232 local structural rules, which are then summarized to a rule corpus of 2,939 local structural rules using their LSR keys. Janus's summarized rule corpus presents a more balanced mix of rule types compared to the raw collection of LSRs.

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 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 [9] and Yang et al [34].

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 the exact method and an average of 18.72 ($\sigma = 7.06$) minutes for the random approach.[8]

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 4.1) 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 4 illustrates the distribution of LSR types after summarization.

6.2 Performance

Next we evaluate Janus' ability to improve pipelines through repairs. Table 1a shows the fraction of pipelines where the candidate repair improved on 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 1b shows the fraction of pipelines where the candidate repair hurts performance. Janus repairs hurt 4% – 39% of pipelines, less than 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 shows the ECDF for the score change in pipelines that improved (Figure 5a) or were hurt (Figure 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.[9]

6.3 Repair distance

Figure 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

[8]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.

[9]https://anonymous.4open.science/repository/67e55703-978b-4977-9b3e-3b9ab9d4f8e8/janus-repairs-formatted.json

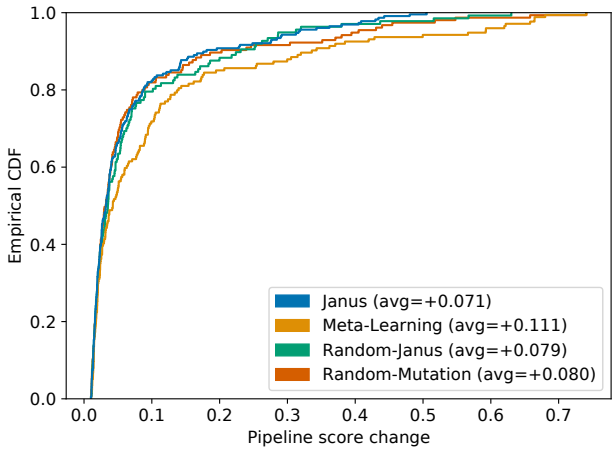
dataset	Janus	Meta-Learning	Random-Janus	Random-Mutation
Hill_Valley_with_noise	0.19 (0.12-0.26)	**0.30 (0.21-0.37)**	0.13 (0.07-0.18)	0.16 (0.09-0.22)
Hill_Valley_without_noise	0.16 (0.10-0.22)	**0.20 (0.13-0.26)**	0.15 (0.09-0.21)	0.08 (0.03-0.12)
breast-cancer-wisconsin	**0.20 (0.14-0.26)**	0.11 (0.06-0.15)	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

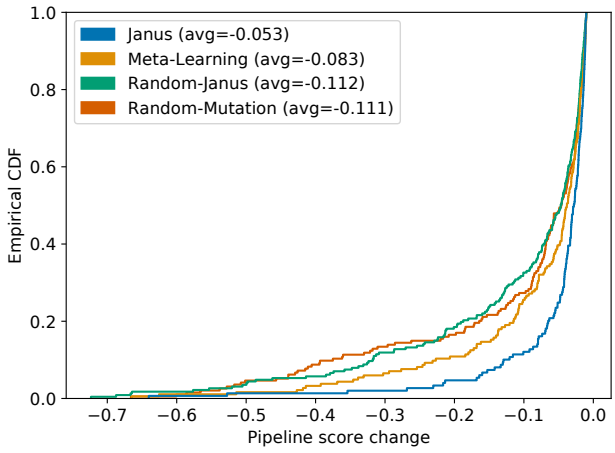
dataset	Janus	Meta-Learning	Random-Janus	Random-Mutation
Hill_Valley_with_noise	**0.11 (0.06-0.16)**	0.16 (0.10-0.21)	0.20 (0.13-0.28)	0.20 (0.13-0.27)
Hill_Valley_without_noise	**0.04 (0.01-0.07)**	0.17 (0.11-0.23)	0.11 (0.06-0.16)	0.21 (0.14-0.27)
breast-cancer-wisconsin	**0.04 (0.01-0.07)**	0.18 (0.12-0.24)	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 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: 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*.

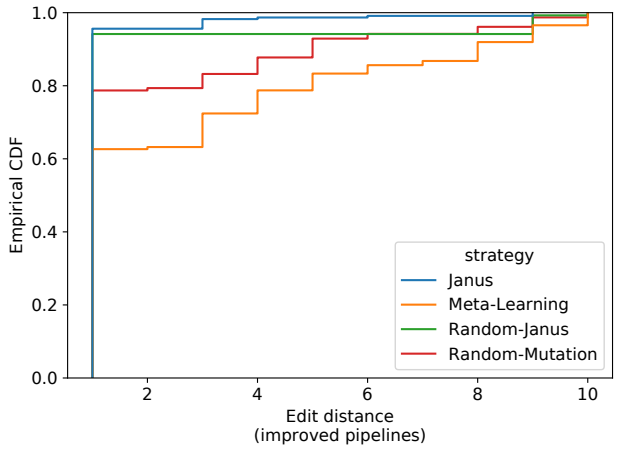


Figure 6: Approaches that use Janus-mined rules (Janus and *Janus-Random*) produce slightly closer repairs compared to *Meta-Learning* and *Random-Mutation*.

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).[10] Hyperparameter-related rules produce an edit distance of 1, which drives Janus repair distances down.

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 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`).[11] The rest of the setup is

[10] Component update rules were a neglible fraction.
[11] This method loosely reduces to a version of TPOT that does not use any genetic programming and produces only sequential pipelines.

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 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.

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 2 shows that with this corpus 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.

7 THREATS TO VALIDITY

Our experiments rely on a corpus of pipelines produced by an automated tool (TPOT [19]). 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 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 [13], 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 pipelines in Janus's target library Scikit-Learn have less then 4 components [22].

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.

8 RELATED WORK

We discuss related work in machine learning, automated program repair, and rule mining in software engineering.

Machine Learning. Recent work has explored a variety of techniques to automatically generate machine learning pipelines. These approaches have encompassed genetic programming [19], Bayesian optimization [6, 27], multi-armed bandit optimization [27], active learning and tensor completion [32, 33], and dynamic program analysis [3]. In contrast to this line of work, Janus does not automatically generate machine learning pipelines. Instead, Janus is focused on applying transformations of an *existing* pipeline to produce a *nearby* pipeline with higher predictive performance. In particular, Janus does not carry out a standard generate-and-evaluate loop, as do many AutoML tools. In our evaluation, Janus uses an AutoML tool, TPOT, to produce a large number of pipelines from which to mine transformations, however, Janus could replace this corpus with pipelines collected through other sources such as collaborative machine learning websites (e.g., OpenML [30]).

Learning to learn, termed meta-learning [29], encompasses techniques to exploit information about prior or related machine learning tasks. Janus's mined pipeline repair rules can be viewed as a form of task-independent meta-learning. Often task-independent meta-learning systems start with a portfolio of pre-existing model candidates, and return the top candidate based on offline evaluations on diverse datasets. In contrast, Janus does not have a fixed set of models, but rather has a portfolio of *transformations* that can produce multiple, possibly previously unseen, pipelines.

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. Recent systems have successfully repaired real bugs in large projects written in production languages such as C and Java [23]. These systems employ approaches such as genetic programming [7], model-based patch ranking [16], patch learning from example changes [2, 14], fix clustering [2], and tree-based edit scripts and patterns [12, 24, 25].

Like prior work [2, 24, 25], Janus treats machine learning pipelines as trees and mines transformations. In contrast to these approaches, Janus is scoped to machine learning pipelines and augments its edit operations with semantic checks unique to machine learning pipelines. To abstract the collection of raw edit rules, Janus uses a simple and effective heuristic (based on LSR keys), rather than deduction-based approaches like template learning [24, 25] or anti-unification [2].

9 CONCLUSION

We frame 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.

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 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

[4] 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.

[5] 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.

[6] 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.

[7] 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

[8] 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

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

[10] 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.

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

[12] 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.

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

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

[23] 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.

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

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

[26] 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).

[27] 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

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

[29] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).

[30] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15, 2 (2013), 49–60. https://doi.org/10.1145/2641190.2641198

[31] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.* 41, 4 (2018), 16–25.

[32] 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.

[33] 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.

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