Automatically Debugging AutoML Pipelines using Maro: ML Automated Remediation Oracle

Julian Dolby
IBM Research
USA
dolby@us.ibm.com

Jason Tsay
IBM Research
USA
jason.tsay@ibm.com

Martin Hirzel
IBM Research
USA
hirzel@us.ibm.com

Abstract

Machine learning in practice often involves complex pipelines for data cleansing, feature engineering, preprocessing, and prediction. These pipelines are composed of operators, which have to be correctly connected and whose hyperparameters must be correctly configured. Unfortunately, it is quite common for certain combinations of datasets, operators, or hyperparameters to cause failures. Diagnosing and fixing those failures is tedious and error-prone and can seriously derail a data scientist's workflow. This paper describes an approach for automatically debugging an ML pipeline, explaining the failures, and producing a remediation. We implemented our approach, which builds on a combination of AutoML and SMT, in a tool called Maro. Maro works seamlessly with the familiar data science ecosystem including Python, Jupyter notebooks, scikit-learn, and AutoML tools such as Hyperopt. We empirically evaluate our tool and find that for most cases, a single remediation automatically fixes errors, produces no additional faults, and does not significantly impact optimal accuracy nor time to convergence.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Error handling and recovery**.

Keywords: AI Debugging, AutoML, Automated Remediation, Automated Debugging

ACM Reference Format:
Julian Dolby, Jason Tsay, and Martin Hirzel. 2022. Automatically Debugging AutoML Pipelines using Maro: ML Automated Remediation Oracle. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS '22), June 13, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3520312.3534868

MAPS '22, June 13, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9273-0/22/06…$15.00
https://doi.org/10.1145/3520312.3534868

1 Introduction

Artificial Intelligence (AI) is an exciting rising paradigm of software development that however also comes with many new challenges for developers. Challenges range from systemic issues such as a lack of education and training [1] and difficulty in reproducibility [13] to hidden technical debt [28] to a need for fairness and controlling for bias [7]. Individual AI developers developing software that trains machine learning (ML) models face tasks covering a wide range from data collection and cleaning to feature selection to training and evaluating models. These tasks are often highly entangled, where errors in earlier tasks often have serious or insidious cross-cutting consequences [16]. Consequences of errors span a wide range depending on the components that they affect, from hard faults to data corruption to incorrect or unintended functionality in the AI system [19]. Similarly, the potential causes of errors are numerous, from the dataset used, derived features, hyperparameters, operators, etc. This complexity in reasoning and tracking errors in AI systems makes them difficult for AI developers to debug.

This paper focuses on the task of debugging a set of possible ML pipelines for a given dataset. Following the terminology of scikit-learn [26], a popular ML framework, we define an ML *pipeline* as a graph of operators and their hyperparameters. Once trained, an ML pipeline becomes an ML model that supports evaluation using metrics and predictions on new unseen data. For this work, we consider *planned pipelines*, which specify a graph of ML operators and schemas for hyperparameters, but leave some choices open, such as concrete hyperparameter settings, or picking one of a choice of multiple operators at a given pipeline step. Given a planned pipeline, a *pipeline instance* fills in all the choices, by picking operators from the set of available options and hyperparameter values from the domain of the corresponding schema. It is common practice to use an automated machine learning (*AutoML*) tool to explore the search space of choices in a planned pipeline to find the best pipeline instance for a given dataset. A pipeline instance is trainable, and can thus be turned into a model and evaluated against metrics for a given dataset. An AutoML search generates and evaluates multiple pipeline instances.

We focus on debugging these planned pipelines because errors in them often propagate to the derived models. Additionally, the automated search often tries erroneous combinations of operators and hyperparameters, which is wasteful. Debugging the failures of a particular ML pipeline is difficult and time-consuming due to the experimental nature of AI development along with the multitude of possible failure causes [3]. Often, the lack of transparency and explainability in AI development results in developers treating pipelines as "black boxes," forcing a trial-and-error approach of testing by running models repeatedly [16]. This is combined with a difficulty of localizing the error due to entanglement or hidden feedback loops [28]. Rather than reason about the development process as a whole with all of its complexities when debugging, our tool embraces the iterative nature of AI development to more efficiently find and remediate bugs.

Our approach combines automated machine learning (AutoML) with a satisfiability modulo theories (SMT) solver to generate, analyze, and remediate instances of a planned ML pipeline for a given task. The complexity and sheer amount of possible causes of failure makes manual debugging difficult [25]. With AutoML, the amount of experiments to reason across when debugging only increases. Our system eases this burden on the AI developer by viewing debugging as a search for constraints over a given space of operators and their hyperparameters, which is a natural fit for an SMT solver. Thus, our system attempts to automatically determine which constraints of operators or hyperparameters prevent certain failures. By using these constraints and the original planned ML pipeline, we generate a remediated planned pipeline that avoids (a generalization of) these failures.

This paper presents a tool named Maro (ML Automated Remediation Oracle) that automatically debugs ML pipelines and generates remediated pipelines based on AutoML experiment results. We build on top of a Python-based open source AutoML interface named Lale [5, 6] that supports composing operators from popular ML libraries such as scikit-learn [26] into pipelines and then running AutoML optimizers such as Hyperopt [8] across these pipelines. Given a user's ML pipeline and their initial AutoML-generated experiments, if some of the experiments have failed, then Maro automatically returns a remediated pipeline. Our tool also provides explanations for the automated remediations for the given ML pipeline through rendering the constraints found by the solver in natural language, as well as displaying the differences between the original and remediated pipelines. We evaluate Maro on 20 planned pipelines that cover a diverse set of ML operators, failure causes, and remediation requirements. We compare Maro against approaches from the BugDoc pipeline debugger [23]. Since BugDoc does not provide remediation, we extend it with this feature to enable better experimental comparison. To the best of our knowledge, our tool is the first to provide a full debugging and remediation round-trip.

The contributions of Maro are as follows:

1. An approach for automated fault localization in ML pipelines based on AutoML and SMT solvers.
2. Automated remediation for ML pipelines, by applying constraints found by the localizer to the original pipeline.
3. Explanation of remediations via natural language as well as via differencing the original and remediated pipeline.

2 Overview and Examples

This section uses examples to give a high-level description of our tool. The target persona is Dante, a data scientist. Dante uses popular Python machine-learning libraries from a Jupyter notebook to build predictive models.

2.1 Detailed Example

Our first example starts when Dante has already inspected the data and found that it has some missing values, categorical features, and discrete target labels. So he assembles a `planned` pipeline with three steps: a `SimpleImputer` for filling in missing values, a `OneHotEncoder` for transforming categoricals into numbers, and a `LogisticRegression` classifier for predicting target labels. The pipe combinator (`>>`) connects operators with dataflow edges, creating a pipeline.

```
one_hot_encoder = OneHotEncoder(handle_unknown="ignore")
planned = SimpleImputer >> one_hot_encoder >> LogisticRegression
```

Dante's day-to-day workflow involves trial-and-error with different pipelines to find the best-performing one. Rather than doing all experiments by hand, Dante uses AutoML tools to automate some of that search. In the example, both `SimpleImputer` and `LogisticRegression` have hyperparameters that Dante deliberately left unspecified. Instead, he uses `Hyperopt` [8] to search possible configurations for them, based on hyperparameter schemas specified in the library. Each evaluation picks a pipeline instance (a pipeline where all hyperparameters are bound to values drawn from their schema) and evaluates it using cross-validation.

```
hyperopt_trainable = Hyperopt(estimator=planned, max_evals=20)
hyperopt_trained = hyperopt_trainable.fit(train_X, train_y)
```

```
Please wait ...
Done, 15 out of 20 evaluations failed, call summary() for details.
```

Unfortunately, most evaluations failed, i.e., the corresponding pipeline instance raised an exception. Dante wonders what he should do now. He is tempted to just ignore the failures and move on, but what if there are root causes that he should understand to build a better pipeline? Given how many evaluations failed, the search may be less effective, as it covered less ground. Moreover, the failures do not come for free: they may have wasted computational resources before raising their exceptions. So rather than give in to the temptation, he decides to poke around a bit. But that prospect fills him with dread: it can become a time drain, since comparing

even a moderate number of pipeline instances (like 20 in this example) is tedious. For now, Dante decides to at least call the `summary()` method as suggested by the error message.

```
hyperopt_trained.summary()
```

name	tid	loss	time	log_loss	status
p0	0	NaN	NaN	NaN	fail
p1	1	-0.514286	0.56447	0.815479	ok
p2	2	-0.514286	0.58215	0.810590	ok
...	...	...	...	...	...
p17	17	NaN	NaN	NaN	fail
p18	18	NaN	NaN	NaN	fail
p19	19	NaN	NaN	NaN	fail

20 rows × 5 columns

Each evaluation in the summary has a name, ID, loss (in this case accuracy, negated to make it a minimization problem), log-loss, and status. Dante decides to retrieve one of the failing instances and pretty-print it as Python code.

```
hyperopt_trained.get_pipeline("p0").pretty_print()
```

```
simple_imputer = SimpleImputer(strategy="median")
one_hot_encoder = OneHotEncoder(handle_unknown="ignore")
logistic_regression = LogisticRegression(
    dual=True,
    fit_intercept=False,
    intercept_scaling=0.48518719297596336,
    max_iter=326,
    solver="liblinear",
    tol=0.006373368408152854,
)
pipeline = simple_imputer >> one_hot_encoder >> logistic_regression
```

As expected, `Hyperopt` chose concrete hyperparameters. But what went wrong? Dante could now look at all the other pipeline instances to find out which choices cause failures. Or he could try to train them and wade through their exception back-traces. Instead, Dante asks Maro, the tool introduced by this paper, for guidance. Maro has three parts: a fault localizer, a remediator, and an explainer. The `auto_remediate()` function first calls the fault localizer and the remediator, taking the original planned pipeline and the evaluations from the Hyperopt run (pipeline instances and their status) and returning a new `remediated` pipeline. The `remediated` pipeline is as similar as possible to the original `planned` pipeline while ruling out all failures observed in earlier evaluations. Lastly, the *explainer* returns a natural language explanation of the suggested remediation.

```
remediated = auto_remediate(planned, hyperopt_trained,
  explanation=True)
```

```
Try setting argument 'strategy' in operator SimpleImputer to 'most_frequent'
```

The explanation pinpoints the cause of failure: `SimpleImputer` should use the `"most_frequent"` strategy. This makes sense, since the dataset is categorical, and other imputation strategies (such as `"median"`) require numeric data. Dante is relieved that Maro guided him to a solution, and decides to try out the remediated pipeline. The remediated pipeline is again a

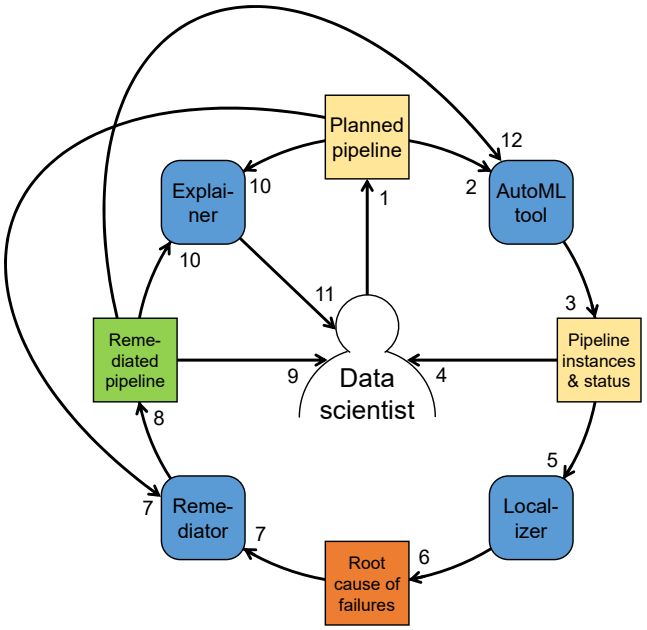

Figure 1. Iterative ML development with Maro.

planned pipeline for which Hyperopt tries pipeline instances by searching the remaining hyperparameters.

```
hyperopt_trainable = Hyperopt(estimator=remediated, max_evals=20)
hyperopt_trained = hyperopt_trainable.fit(train_X, train_y)
```

```
Please wait ...
Done, all evaluations succeeded.
```

This time, all 20 out of 20 evaluations succeeded. So Dante can get back to his work of finding the best pipeline for the dataset. He can evaluate the pipeline on test data, or perhaps use AutoML to search different classifier choices.

2.2 Tool Overview

Figure 1 gives an overview of how a data scientist such as Dante can use our tool Maro. The workflow starts with the data scientist, shown in the center, creating a planned pipeline (1). They can then feed this pipeline to an AutoML tool, such as grid-search, Hyperopt, or any other backends that Lale supports (2). The automated search yields a set of pipeline instances along with their status, which can be "ok" or "fail" (3). Without Maro, the data scientist would have little choice but to manually inspect these results (4). But a better option is to send the results on to Maro's fault localizer component (5). The localizer uses an SMT solver to find a root cause of the failures (6). This root cause, along with the original planned pipeline, forms the input to Maro's remediator component (7). The result is a remediated pipeline (8), which the data scientist can inspect directly if they so wish (9). Alternatively, to make the fix easier to understand, the data scientist can send the remediated pipeline and the original pipeline to Maro's explainer component (10). This explains the remediation to the data scientist by rendering it in natural language (11). And finally, as the remediated pipeline is

itself a planned pipeline, the data scientist can use it as input to the AutoML tool (12), thus completing the circle.

2.3 Additional Use Cases

Maro can handle a diverse set of ML pipelines and associated failures. This paper experiments with a set of 20 planned pipelines. We initially chose a set of pipelines based on interviewing ML practitioners and analyzing publicly-available pipelines. Then, we grew that set as we implemented and tested Maro to exercise challenging corner cases. All planned pipelines use common ML operators, mostly from scikit-learn [26], such as `LogisticRegression`, or `OneHotEncoder`, but also operators from other scikit-learn compatible libraries, such as a bias mitigator from AIF360 [7] and gradient-boosted trees from LightGBM [21]. The full list of pipelines is available in the extended version of this paper [10].

The pipelines failed for a variety of reasons, including characteristics of the input data; incompatible operators; incompatible hyperparameters; or some combination of the above. Some pipelines failed fast, others only after expensive training of a prefix. Sometimes, even hyperparameters within a single operator can be incompatible with each other. This is known as a conditional hyperparameter constraint, and some AutoML tools prune invalid combinations from the search space based on manual specification, e.g., auto-sklearn [11] or Lale [6]. However, other AutoML tools do not come with comprehensive conditional hyperparameter constraint specifications, e.g., scikit-learn's GridSearchCV.

Maro repairs each planned pipeline in our set to prune failing instances from the search space. Remediations may involve removing operators from choice for AutoML algorithm selection; limiting categorical hyperparameters to a set of values (such as the complement of removing a value from an enum); placing upper or lower bounds on continuous hyperparameters; or some combination of the above. While data is sometimes but not always part of the problem, remediation is always in the pipeline, not in the data. This is because in practice, data scientists must work with the data at hand. Thankfully, often, the purpose of an operator is to transform the data you have into the data you need, so picking and configuring operators in a pipeline can also fix data problems.

3 Algorithms and Tool Design

As shown in Figure 1, Maro has three main components:

1. A *localizer* that, given a set of evaluations, computes a root cause of failures, i.e., operator choices and hyperparameter settings that correlate with pipeline instances that failed.
2. A *remediator* that, given the original planned pipeline and the root cause of failures, constructs a new pipeline that excludes known failures while allowing other settings.

```
one_hot_encoder = OneHotEncoder(handle_unknown='ignore')
ordinal_encoder = OrdinalEncoder(handle_unknown='ignore')
encoder_choice = one_hot_encoder | ordinal_encoder
planned = (project_categoricals >> encoder_choice
          >> StandardScaler >> LogisticRegression)
```

Figure 2. Example pipeline (k).

if H_p(`StandardScaler.with_mean`) = `False`
then `True`
else H_p(`OrdinalEncoder.handle_unknown`) = `"ignore"`

Figure 3. Localizer-generated constraint for pipeline (k).

3. An *explainer* that, given the original planned pipeline and the root cause of failures, computes an explanation that makes the remediation easier to understand.

We start with some preliminaries and defining Maro's interfaces, then present how the three main components work.

3.1 Preliminaries

The input to Maro consists of a set of evaluations, which are pipeline instances along with their status and the original planned pipeline.

Definition 3.1 (Pipeline)**.** A planned *pipeline* P is a set of steps $S_0, \ldots, S_n$, which are operators or operator choices.

Definition 3.2 (Pipeline instance)**.** A *pipeline instance* p is a pipeline along with a Boolean result r_p denoting success or failure and a mapping H_p from hyperparameters to values.

To simplify the discussion, we model operator choice for algorithm selection by the presence of a hyperparameter that identifies the chosen operator.

Constraints. Maro uses an interface of *constraints* to communicate between the fault localizer and the remediator: the localizer computes constraints that capture successful runs, and the remediator alters the initial planned pipeline to rule out pipeline instances that violate those constraints.

There are two kinds of constraints, atomic and multiple. An atomic constraint compares a hyperparameter against a constant (e.g., H_p(`SimpleImputer.strategy`) ≠ `"median"`) or against another hyperparameter or checks if a hyperparameter is present. A multiple constraint arranges other constraints in an if-then-else tree. To make this concrete, consider Figure 3, which shows the constraints our solver found for example pipeline (k), shown in Figure 2. The if-part represents the top of the tree, checking whether `StandardScaler.with_mean` is `False`. The then-clause is simply `True`, indicating that the pipeline is valid. The else-clause says that otherwise, the pipeline is valid if `OrdinalEncoder.ignore_unknown` is present and set to `"ignore"`, implying that the operator choice picked `OrdinalEncoder`.

3.2 Fault Localization

Maro receives a set of pipeline instances $\mathfrak{P}, p_1, \ldots, p_n$, and computes hyperparameter constraints C in the format of Section 3.1 that determine if a pipeline instance fails. To allow the most flexibility in determining the constraints while

also ensuring that remediation is feasible, our approach uses templates of constraints we can handle but these templates are made flexible with symbolic variables that control the specific constraints.

To do this, Maro uses the solver-aided language Rosette [30]. Solver-aided languages allow programming with symbolic values. Intuitively, symbolic values can be used for any program value (of a supported type), and the result of running such a program is a logical formula that, when solved, yields concrete values for the given symbolic ones such that the program succeeds. This allows us to write logic that checks whether a given constraint explains all failures, leaving the actual constraint symbolic so that the solver fills it in.

Atomic Constraint. To see how this works, consider the example from Section 2.1, where `SimpleImputer` with hyperparameter `strategy` set to `"median"` breaks on non-numeric data. This is an atomic constraint that invalidates pipelines, which is the simplest case. If we somehow knew the constraint to use, we could write the following:

$$E(\mathfrak{P}) \equiv \forall_{p \in \mathfrak{P}} \left(\begin{array}{l} r_p \iff \\ H_p(\texttt{SimpleImputer.strategy}) \neq \texttt{"median"} \end{array} \right)$$

This formula states that a pipeline instance from $\mathfrak{P}$ succeeds if and only if it does not bind `SimpleImputer.strategy` to `"median"`. If we think of E as instrumenting execution of AutoML, so it sees all attempted pipelines and their outcomes, it will be true for the example from Section 2.1, since those pipelines indeed fail in precisely that case. This would be simple to do; however, we do not, in general, know in advance what hyperparameter to check. But symbolic variables—denoted by @ in Rosette—allow us to leave the actual constraint unspecified and have the solver fill it in. We can write that as follows:

$$S_1(\mathfrak{P}) \equiv \forall_{p \in \mathfrak{P}} \left(r_p \iff H_p(@hparam) = @value \right)$$

The process of abstractly executing the symbolic program plays the role of the instrumentation mentioned above: the solver at the end finds a binding of the symbolic variables that make execution valid, if such there be. Thus it binds the symbolic variables $@hparam$ and $@value$ to concrete values that make the assertion true. This will find any hyperparameter setting that correlates exactly with pipelines that fail. In fact, this simple logic suffices for any failure caused by a single value of a single hyperparameter. The symbolic variables can be read directly from the solution to generate atomic constraints as described in Section 3.1.

There are several categories of error that similar constraints can capture (discussed in more detail in [10]). They are the following (S_2, S_3, and S_4):

- presence of a hyperparameter, regardless of value
- numerical restriction to be more or less than a given value
- numerical constraints between hyperparameters

Multiple Constraints. While in some cases a single atomic constraint suffices, that is not always the case. Consider the example pipeline (k), in which the combination of `with_mean` for `StandardScaler` and `handle_unknown` for `OneHotEncoder` breaks for this dataset. Either is allowed, but they cannot be used together. To handle this, we stack these constraints such that one constraint controls which other constraint applies; the superscripts on S indicate that the three uses of S generate distinct symbolic variables, so there are three independent constraints:

$$S_5 \equiv \forall_{p \in \mathfrak{P}} \left(\begin{array}{l} r_p \iff \\ \text{if } S^1_{any}(\{p\}) \text{ then } S^2_{any}(\{p\}) \text{ else } S^3_{any}(\{p\}) \end{array} \right)$$

This is a tree structure of the constraints, and the values for all the constraints can be read directly from the variables produced by the solver. This only illustrates two levels, but clearly they can be stacked as deeply as needed. The localizer communicates its results to the remediator by providing the symbolic constraint C_i of each S^i in the format described in Section 3.1, as exemplified in Figure 3.

3.3 Remediation

The remediator computes a remediated planned pipeline corresponding to the formula $origPipe \wedge C$, describing a set of possible pipeline instances for AutoML to sample from. Here, *origPipe* is a formula that describes the original planned pipeline. It characterizes a (usually unbounded) set of possible pipeline instances from which the initial AutoML run sampled a finite set of instances. And C is a formula returned by the localizer that rules out a generalization of the concrete failed instances, abstracted to be brief and broadly applicable.

As discussed in Section 3.1, the C formula can involve if-then-else, expressible via negation, conjunction, and disjunction. Hence, one approach would be to perform remediation in a purely logical representation and then, only at the end, convert back to a pipeline representation suitable for AutoML tools. Unfortunately, this would make the result of remediation inscrutable for data scientists, since it may look nothing like *origPipe*. Therefore, for the sake of better explainability, Maro's remediation algorithm takes a bottom-up approach of directly constructing a remediated pipeline that resembles *origPipe*.

Figure 4 shows Maro's remediation algorithm. As described in Section 3.1, the solver returns constraints arranged as a tree, encoding conditionals where the parent is an if-clause and subtrees represent then and else clauses. Lines 2–5 handle this case by recursive remediation calls for the left and right subtree. The `makeChoice` function combines the results via Lale's choice combinator (|). When the algorithm reaches a leaf, it faces a conjunction constraint, handled by Lines 6–8 via recursive calls to remediate conjuncts one by one.

The base case of the recursion, in Figure 4 Line 9, is a (possibly negated) atomic constraint. Lines 10–13 determine which operators are included in the remediated pipeline. If a constraint notes that an operator's hyperparameter must be *present* or cannot be *absent* and the corresponding operator

```
1  algorithm process(origPipe, C):
2      case C ≡ (if C1 then C2 else C3):
3          thenPipe = process(origPipe, C1 ∧ C2)
4          elsePipe = process(origPipe, ¬C1 ∧ C3)
5          return makeChoice(thenPipe, elsePipe)
6      case C ≡ C1 ∧ C2:
7          leftPipe = process(origPipe, C1)
8          return process(leftPipe, C2)
9      case isAtomicConstraint(C):
10         if affectsPresenceOfOperators(C):
11             tmpPipe = restrictChoice(origPipe, C)
12         else:
13             tmpPipe = origPipe
14         if comparesMultipleHyperparameters(C):
15             return makeComparison(tmpPipe, C)
16         else:
17             return customizeSchemas(tmpPipe, C)
```

Figure 4. Pseudo-code for Maro's remediation algorithm.

```
pca = PCA.customize_schema(n_components=features_schema)
select_k_best = SelectKBest.customize_schema(k=features_schema)
planned = pca >> select_k_best >> LogisticRegression
```

Figure 5. Example pipeline (g).

is part of a choice, `restrictChoice` removes that choice from the pipeline in favor of the required operator.

Line 14 detects whether the constraint involves multiple hyperparameters (possibly from multiple operators), such as in pipeline (g) in Figure 5, where `PCA.n_components` must be less than `SelectKBest.k` because otherwise too few columns would be piped to `SelectKBest`. In these cases, because schemas are modularized per-operator, and because JSON schema cannot express a less-than constraint involving two hyperparameters, function `makeComparison` in Line 15 proxies this constraint by splitting the possible values for the non-dependent hyperparameter into a number of ranges (our default is five). For example, if `k` can range from 5..55, then five versions of the `SelectKBest` operator are created where `k` may range from 5..15, 16..25, ..., 46..55. Then, the dependent hyperparameter is also split such that it complies to the constraint. For example, if `n_components` originally ranged from 1..40, then five versions of `PCA` are created where `n_components` may range from 1..4, 1..15, ..., 1..40, thus guaranteeing that it is less than the corresponding `k` range. Finally, `makeComparison` combines these pairs via Lale's choice combinator (`|`).

Lastly, Line 17 handles the simplest and most common case of applying constraints to a single hyperparameter and operator. Constraints may either limit a hyperparameter to a set of values or, if negated, exclude them from a given set of values. To apply such constraints, we use Lale's `customize_schema` feature, which returns a copy of an operator that specifies a different schema for one of its hyperparameters. Recent work shows how to make JSON Schema closed under conjunction and negation [4], but since that work is not open-source, we implemented our own. We translate a given constraint into the corresponding schema, as in the example in Section 2.1 that restricts the `strategy` hyperparameter of the `SimpleImputer` operator to the value `"most_frequent"`.

Maro's remediator is flexible enough to be used with other localization algorithms so long as they output constraints in a compatible format. We implemented alternative algorithms and successfully used them with our remediator as part of our evaluation, as described in more detail in Section 4.2.

3.4 Explanation

The final component of Maro is an explainer that assists the user in understanding the suggested remediation found by the solver via natural language. Similar to the remediator, Maro's explanation features are flexible enough to be used with other localization methods as long as they output constraints in a compatible format.

Creating a natural language explanation uses a similar algorithm as that for remediation in Figure 4. The main difference is that the helper functions `makeChoice`, `restrictChoice`, `makeComparison`, and `customizeSchema` generate natural language instead of Python code. For instance, `makeChoice` for explanation simply joins constraints using the English word "OR" and newlines. The other difference is that Line 8, instead of making a chained call on the output of the previous step, uses the English word "and." For a full example, consider the explanation for example pipeline (k):

```
Try setting argument 'with_mean' in operator StandardScaler to 'False'
OR
Try setting argument 'with_mean' in operator StandardScaler to 'True'
 and try ensuring that argument 'handle_unknown' in operator OrdinalEncoder
 is present for all runs (a Choice operator may need to be removed)
```

4 Evaluation

This section presents experiments for three research questions:

RQ1: How does Maro's remediation affect correctness compared to baseline approaches?
RQ2: How does Maro's remediation affect accuracy?
RQ3: Does Maro's remediation help converge to optimal configurations more quickly?

We include the pipelines, results, and version of Maro used in this evaluation as part of the replication kit[1].

4.1 Baseline Localization Algorithms

We compare the correctness of Maro to other baseline ML fault localization algorithms: modified versions of the *Shortcut* and *Stacked Shortcut* methods from BugDoc [23]. These algorithms only attempt to identify and report root causes for failures as constraints and do not include remediation, so we convert the reported root causes into a compatible format for Maro's remediator and explainer.

4.2 Correctness (RQ1)

Maro is inherently a correctness tool: given a set of evaluations, some of which are incorrect, it locates the fault and repairs the planned pipeline. However, when a new AutoML

[1] https://zenodo.org/record/6385800

Table 1. Correctness evaluation per localization method.

Localization	Successful	Restrictive	Unsuccessful
Maro	17	5	3
Shortcut	7	1	13
Stacked Shortcut	7	2	13

search is launched starting from the remediated planned pipeline, it will almost certainly attempt new pipeline instances that Maro has not seen before. There is no a priori guarantee that those new instances do not fail in new ways.

Our evaluation set is the set of 20 pipeline use-cases described in Section 2.3 which cover a wide variety of potential failure cases. To create this set, we started with problematic planned pipelines mined from OpenML, plus data scientist interviews mentioning common failure causes. After that, one author created additional cases to challenge our tool, drawing upon documented constraints, Python raise statements, and reported issues. For each example pipeline, we report whether each method was able to find a remediation and whether failures occured after 20 more AutoML-generated evaluations based on the remediation.

Table 1 summarizes the results. A remediation is considered successful if it generates no failures after 20 more AutoML-generated evaluations based on the remediation. Maro is able to successfully remediate all but three cases, whereas the baseline methods are only able to successfully determine root causes in seven cases each. (Four of these successful remediations are due to examples that require removing operators from choices which are part of our modifications. Without such modifications, the number of successes would be lower.) In five examples, Maro suggests a fix that is more restrictive but does not generate failures. A restrictive remediation is one that may restrict the potential search space for an AutoML pipeline more than a manual remediation. This may be due to a limitation of this evaluation method where Maro only has access to 20 automatically-generated examples which may insufficiently cover the space of expected fixes. For example, an ideal remediation for a pipeline may be a constraint where `n_neighbors`$\leq$15. With an input of 20 evaluations, Maro suggested a constraint of $\leq$8, which is more restrictive but technically correct. Increasing the input to 50 evaluations increased the constraint to $\leq$13, which is closer to the ideal remediation. We expect that both the restrictive and unsuccessful remediations might be improved with additional input evaluations or a second round of remediation. We note that because Maro supports a full round-trip, we are able to perform successive automated debugging on unsuccessful remediated pipelines.

The baseline methods insufficiently find root causes for a number of reasons. One reason is that they are simply not expressive enough to successfully remediate the pipeline. One example is example pipeline (k) as seen in Figure 2, where `with_mean` only has a constraint depending on the encoder selected. The baseline methods only express simple equality or inequality constraints. Simply reporting a single constraint or even a union of constraints is insufficient to describe this remediation. Another reason is that the baseline methods assume that hyperparameters are independent and can be freely swapped without additional consequences. However, hyperparameters are sometimes dependent on each other even across operators, such as in pipeline (g) in Figure 5, where `k` must be $\leq$`n_components`. Lastly, the baseline methods each only consider a single failing pipeline instance whereas Maro considers all failing instances. Although we did not implement the *Debugging Decision Tree* method from BugDoc [23], we expect that if we modified it in similar ways to the *Shortcut* method and augmented it with our automated remediation, it would fail to find remediations for many of the cases for similar reasons. It is also highly expensive (exponential time) to run for a realistic ML pipeline so we chose not to reimplement it, especially given that we expect similar performance to *Shortcut*.

4.3 Accuracy (RQ2) and Convergence (RQ3)

Although Maro focuses on correctness, that must be balanced with predictive performance on the given dataset. Since Maro is designed to work with AutoML tools, it is possible that the remediation may remove too much of the potential search space in order to guarantee a correct pipeline. We would then expect new AutoML searches on this remediated pipeline to also perform poorly. We compare the AutoML predictive performance of the original pipeline to that of the remediated pipeline provided by Maro in terms of test set accuracy and number of iterations.

For the accuracy evaluations, we run two AutoML jobs: the original pipeline and the remediated pipeline returned by Maro after 20 evaluations of the first job. We use a train+test split of 80%+20% for the given dataset and run each AutoML job for 1,000 iterations for both the original and remediated pipeline using Hyperopt [8]. Let "optimal" accuracy refer to the best test set accuracy discovered in 1,000 iterations [2]. We run each job five times and report the average optimal accuracy discovered by the five identical AutoML jobs and the average number of iterations taken to reach it.

Remediated pipelines have better optimal accuracy in 8 out of 20 cases and the same optimal accuracy in an additional 7 out of 20 cases, while the original pipeline has better accuracy in the remaining five cases. For the cases with differing accuracies, the average difference is relatively small at 0.0049, and a paired t-test suggests that original and remediated optimal accuracies do not vary significantly (p=0.149). For RQ2, this suggests that the remediations created by Maro on average do not reduce accuracy and therefore are not removing potentially beneficial sections of the search space.

We also examine the number of iterations to reach optimal accuracy, specifically for the 15 cases where the remediated pipeline discovers a better or equal optimal accuracy than the original. We focus on these cases to have a similar point

of comparison in terms of iterations needed. We compare the average number of iterations needed for the remediated pipeline to match or surpass the average optimal accuracy of the original. In these cases, more remediated pipelines reach the original pipeline's optimum accuracy faster (10 out of 15). The original pipeline is faster in four cases while in one case both reach the optimal accuracy in the same average number of iterations. However, a paired t-test suggests that the average iterations for original and remediated pipelines to discover the original optimal accuracy do not vary significantly (p=0.935). For RQ3, this suggests that remediations created by Maro on average also do not change time to convergence compared to original pipelines.

5 Threats to Validity

The biggest limitation might appear to be that we only show remediations based on the five formulae S_1 to S_5 in Section 3. However, these formulae turn out to be sufficient for all 20 planned pipelines used in the evaluation. These formulae are much more expressive than similar tools which are unable to cover all example pipelines. We also note that these formulae could easily be extended for Maro. Another limitation is that our tool does not guarantee finding minimal root causes for all possible instances of a given planned pipeline but only finds a root cause for a set of given instances, usually generated by an AutoML job. However, our experiments (RQ1) suggest that a relatively modest number of instances (20) is enough correctly remediate a pipeline in most cases. In the evaluation of Maro, we did not perform a human user study to examine usability. Although we do consider usability valuable, our central claims of correctness, accuracy, and convergence do not rely on human studies for evaluation.

6 Related Work

AI development and AutoML. Systems with AI components come with unique challenges for the engineering process and individual developers. Debugging in particular is challenging due to errors hiding in data rather than code [17, 31] and the sheer amount of effort involved in manual evaluation due to the potentially millions of parameters to inspect [3, 16, 19]. To reduce this burden of manually exploring ML models, automated machine learning (AutoML) tools such as auto-sklearn [11] and AutoKeras [20] use Bayesian optimizers to automatically construct ML pipelines and their hyperparameters. We position our work among tools that expand the capabilities of AutoML rather than improve the search or optimization performance. One such tool is Lale [5, 6], which is a library of Python interfaces around ML operators designed to provide a consistent method of specifying pipelines for AutoML. Maro takes advantage of Lale's ability to precisely specify the search space for the pipeline's operators and hyperparameters. Another such tool is AMS [9], which automatically "strengthens" weak pipeline specifications for AutoML by providing alternative operators and suggested hyperparameter spaces to search via learning over an existing corpus of AI software.

Data and AI debugging tools. We position our work among other tools that assist in debugging and troubleshooting data-centric and AI software. Data-centric tools such as Panda [18] and PerfDebug [29] use data provenance to aid in debugging data-centric pipelines and post-mortem performance issues respectively. BigSift [12] automatically generates a minimum set of inputs that reproduce a test failure when given an Apache Spark program, test oracle, and input dataset. Dagger [27] is an end-to-end system for debugging data-centric errors in pipelines where users manually instrument their Python code for later logging and querying. AI debugging tools such as LEMON [32] and Haq et al. [15] automatically generate test suites for deep learning frameworks and KP-DNNs respectively. Nushi et al. [25] describe a human-in-the-loop methodology for troubleshooting AI systems which uses crowd-sourcing to simulate potential fixes to components. DialTest [22] is a tool for automatically detecting faults in RNN-driven dialogue systems using transformations guided by Gini impurity. Habib et al. use JSON subschema checks to find bugs in ML pipelines [14]. Though not AI-related, our tool is also related to constraint-based automated program repair tools such as SemFix [24].

The tool closest to our work is BugDoc [23], which automatically infers root causes of failures in ML pipelines based on previous executions. This is similar in concept to Maro's localizer component. BugDoc does not attempt to remediate nor further explain root causes unlike Maro. To our knowledge, Maro is the first tool to implement automated remediation and natural language and visual constraint explanation components in the context of debugging ML pipelines. Our tool's localizer component also differs from BugDoc in that it is designed for planned pipelines that work with AutoML and is also more expressive. Debugging support for such pipelines is more complex than normal ML pipelines in that each pipeline is a search space where operators and parameters may vary. To our knowledge, Maro is the first tool of its kind to express complex constraints between hyperparameters, operators, and other constraints. This is reflected in the evaluation in Section 4.2 where baselines based on BugDoc algorithms are unable to identify root causes in most cases whereas Maro is able to. The aforementioned section details the differences between BugDoc and Maro's localizer component as they relate to our empirical evaluation.

We note that among all of the related automated (i.e. [12, 15, 22, 23]) or end-to-end (i.e. [27]) debugging tools, Maro is unique in that it not only automatically identifies bugs but remediates them as well rather than only identifying root causes or generating test cases. This round-trip from actionable ML pipeline to remediated actionable ML pipeline is, to our knowledge, unique to Maro.

7 Broader Impacts

Automated machine learning (AutoML) in general encourages computationally-heavy approaches to common data science tasks which raise CO_2 emissions. We believe that our tool encourages less computational waste by enabling data scientists to more efficiently use AutoML by not wasting resources on failing combinations of operators and hyperparameters. We also hope that Maro enables data scientists to fix errors faster and run less AutoML jobs overall. However, it is also possible that helping data scientists more easily debug AutoML pipelines may encourage further usage of automated techiniques which may overall raise CO_2 emissions. Future work may explore techniques to reduce the amount of initial AutoML iterations necessary to remediate pipelines in order to encourage less wasteful automated machine learning.

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 *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[3] A Arpteg, B Brinne, L Crnkovic-Friis, and J Bosch. 2018. Software Engineering Challenges of Deep Learning. In *Conference on Software Engineering and Advanced Applications (SEAA)*. 50–59. https://doi.org/10.1109/SEAA.2018.00018

[4] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Conférence sur la Gestion de Données (BDA)*. https://arxiv.org/abs/2104.14828

[5] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. Lale: Consistent Automated Machine Learning. In *KDD Workshop on Automation in Machine Learning (AutoML@KDD)*. https://arxiv.org/abs/2007.01977

[6] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, Avraham Shinnar, and Jason Tsay. 2021. Pipeline Combinators for Gradual AutoML. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf

[7] R K E Bellamy, K Dey, M Hind, S C Hoffman, S Houde, K Kannan, P Lohia, J Martino, S Mehta, A Mojsilović, S Nagar, K N Ramamurthy, J Richards, D Saha, P Sattigeri, M Singh, K R Varshney, and Y Zhang. 2019. AI Fairness 360: An extensible toolkit for detecting and mitigating algorithmic bias. *IBM Journal of Research and Development* 63, 4/5 (jul 2019), 4:1–4:15. https://doi.org/10.1147/JRD.2019.2942287

[8] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (ICML)*. 115–123.

[9] José P. Cambronero, Jürgen Cito, and Martin C. Rinard. 2020. AMS: Generating AutoML Search Spaces from Weak Specifications. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 763–774. https://doi.org/10.1145/3368089.3409700

[10] Julian Dolby, Jason Tsay, and Martin Hirzel. 2022. Automatically Debugging AutoML Pipelines Using Maro: ML Automated Remediation Oracle (Extended Version). arXiv:2205.01311 [cs.SE]

[11] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Conference on Neural Information Processing Systems (NIPS)*. 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf

[12] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 863–866. https://doi.org/10.1145/3236024.3264586

[13] Odd Erik Gundersen and Sigbjørn Kjensmo. 2017. State of the Art: Reproducibility in Artificial Intelligence. In *Conference on Artificial Intelligence (AAAI)*. 1644–1651. https://ojs.aaai.org/index.php/AAAI/article/view/11503

[14] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *International Symposium on Software Testing and Analysis (ISSTA)*. 620–632. https://doi.org/10.1145/3460319.3464796

[15] Fitash Ul Haq, Donghwan Shin, Lionel C Briand, Thomas Stifter, and Jun Wang. 2021. Automatic Test Suite Generation for Key-Points Detection DNNs Using Many-Objective Search (Experience Paper). In *International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 91–102. https://doi.org/10.1145/3460319.3464802

[16] C Hill, R Bellamy, T Erickson, and M Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 162–170. https://doi.org/10.1109/VLHCC.2016.7739680

[17] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *International Conference on Software Engineering (ICSE)*. 1110–1121. https://doi.org/10.1145/3377811.3380395

[18] Robert Ikeda, Junsang Cho, Charlie Fang, Semih Salihoglu, Satoshi Torikai, and Jennifer Widom. 2012. Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows. In *International Conference on Data Engineering (ICDE)*. 1249–1252. https://doi.org/10.1109/ICDE.2012.118

[19] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 510–520. https://doi.org/10.1145/3338906.3338955

[20] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Conference on Knowledge Discovery and Data Mining (KDD)*. 1946–1956. http://doi.acm.org/10.1145/3292500.3330648

[21] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Conference on Neural Information Processing Systems (NIPS)*. 3146–3154. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree

[22] Zixi Liu, Yang Feng, and Zhenyu Chen. 2021. DialTest: Automated Testing for Recurrent-Neural-Network-Driven Dialogue Systems. In *International Symposium on Software Testing and Analysis (ISSTA)*. 115–126. https://doi.org/10.1145/3460319.3464829

[23] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. BugDoc: A System for Debugging Computational Pipelines. In *International Conference on Management of Data (SIGMOD)*. 2733–2736. https://doi.org/10.1145/3318464.3384692

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[25] Besmira Nushi, Ece Kamar, Eric Horvitz, and Donald Kossmann. 2017. On Human Intellect and Machine Failures: Troubleshooting Integrative Machine Learning Systems. In *Conference on Artificial Intelligence (AAAI)*. 1017–1025. https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15032/0

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)* 12 (2011), 2825–2830.

[27] El Kindi Rezig, Ashrita Brahmaroutu, Nesime Tatbul, Mourad Ouzzani, Nan Tang, Timothy Mattson, Samuel Madden, and Michael Stonebraker. 2020. Debugging Large-Scale Data Science Pipelines Using Dagger. In *Demonstration at the Conference on Very Large Data Bases (VLDB-Demo)*. 2993–2996. https://doi.org/10.14778/3415478.3415527

[28] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Conference on Neural Information Processing Systems (NIPS)*. 2503–2511. http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems

[29] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. 2019. PerfDebug: Performance Debugging of Computation Skew in Dataflow Systems. In *Symposium on Cloud Computing (SoCC)*. 465–476. https://doi.org/10.1145/3357223.3362727

[30] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 135–152. https://doi.org/10.1145/2509578.2509586

[31] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. 2019. How does Machine Learning Change Software Development Practices? *Transactions on Software Engineering (TSE)* (2019). https://doi.org/10.1109/TSE.2019.2937083

[32] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2020)*. 788–799. https://doi.org/10.1145/3368089.3409761