Hyperparameter Optimization for AST Differencing

Matias Martinez ⬤, Jean-Rémy Falleri ⬤, and Martin Monperrus ⬤

Abstract—**Computing the differences between two versions of the same program is an essential task for software development and software evolution research. AST differencing is the most advanced way of doing so, and an active research area. Yet, AST differencing algorithms rely on configuration parameters that may have a strong impact on their effectiveness. In this paper, we present a novel approach named** DAT **(*Diff Auto Tuning*) for hyperparameter optimization of AST differencing. We thoroughly state the problem of hyper-configuration for AST differencing. We evaluate our data-driven approach** DAT **to optimize the edit-scripts generated by the state-of-the-art AST differencing algorithm named GumTree in different scenarios.** DAT **is able to find a new configuration for GumTree that improves the edit-scripts in 21.8% of the evaluated cases.**

Index Terms—**Software evolution, Tree differencing, Abstract Syntax Trees (AST), hyperparameter optimization, edit-script.**

I. INTRODUCTION

THE computation of the differences between two versions of the same program is an essential task for software development. It is done on a daily basis when developers share and discuss code changes, through pull-requests in development platforms (e.g. Github, Gitlab, etc.) or through patches over mailing lists (patches on the Linux Kernel Mailing List). The most common differencing strategy of developer tools is line-based: a diff is a list of *chunks* where each chunk is a set of added and/or removed lines. For example, `git` provides the widely used command 'git-diff' for computing a line-based diff script. To push this state-of-practice further, there is an active line of research on differencing algorithms that work at the level of Abstract Syntax Trees (AST) instead of lines [11], [16], [17], [21], [23], [26]. In this case, the differences between two program versions are expressed in terms of actions over nodes from the trees under comparison (e.g., Insert, Remove, Update and Move nodes). AST differencing has been shown to

be significantly better than line differencing for a number of tasks [15], [17].

AST differencing is of great importance not only for practitioners, but also for software engineering research. AST differencing algorithms have been extensively used to study software evolution [20], [29], [51]. Key usages of AST differencing are pattern inference for automated program repair [32], [33], [34], bug fix analysis [37], [49], [55], code recommendation [39], [40], [43]. Consequently, it is of great importance for both practitioners and researchers to have reliable AST differencing tools.

Virtually all AST differencing algorithms can be configured in some way. That is, they have *hyperparameters* that control and guide the differencing process. For example, the state-of-the-art differencing tool GumTree has one hyperparameter called `BUM_SMT` for setting a minimum similarity threshold between two trees. This parameter has a default value of 0.5. Changing that value impacts on the produced edit-scripts.

Unfortunately, as shown in previous research [14], [18], state-of-the-art tools such as GumTree often generate sub-optimal diffs, that is, edit-scripts that contain *spurious* AST operations. *In this paper, we show that these hyperparameters matter and that one can improve AST differencing performance by finding hyperoptimized values.*

In this paper, we present a novel approach named DAT (*Diff Auto Tuning*) for hyperparameter optimization of the AST differencing. Given an AST differencing algorithm, DAT finds the optimal parameter configuration given a set of file-pairs to be compared. DAT requires neither ground-truth nor labeled file pairs, the optimization is guided by a widely accepted quality metric for AST differencing [16], [17], [23], [24], [25], [26], [38]: the length of the edit-script. To our knowledge, we are the first to thoroughly study the problem of suboptimal configuration for AST differencing and to propose a data-driven approach to solve this problem.

We evaluate DAT by searching for the best configurations of the AST differencing algorithm GumTree. In particular, we evaluate it in two scenarios. The first scenario consists of searching for the configuration that works best on a set of file pairs. In this case, DAT does *global hyperparameter optimization*. The configuration found can be used as the new default configuration when a practitioner aims to apply diffs in a new programming language or AST meta-model. Secondly, we evaluate the ability of DAT to find the best configuration for a particular case (i.e., a single file-pair). In this case, DAT does *local hyperparameter optimization*. Our evaluation consists of executing those optimizations on more than 9,000 pairs of Java files extracted from real-world commits of open-source Java projects.

Matias Martinez is with the Universitat Politècnica de Catalunya, CP 08034 Barcelona, Spain (e-mail: matias.martinez@upc.edu).

Jean-Rémy Falleri is with the CNRS, Bordeaux INP, LaBRI, Univ. Bordeaux, UMR 5800, CP F-33400 Talence, France, and also with the Institut Universitaire de France, France (e-mail: jean-remy.falleri@u-bordeaux.fr).

Martin Monperrus is with the KTH Royal Institute of Technology, CP 114 28 Stockholm, Sweden (e-mail: monperrus@kth.se).

The research question that we investigate in this paper are:
- RQ1: To what extent does hyperparameter optimization improve the performance of AST differencing?
- RQ2: To what extent can one speed up hyperoptimization using more advanced techniques than exhaustive grid search?
- RQ3: To what extent is local hyperparameter optimization effective?
- RQ4: What is the overhead of local hyperparameter optimization?

Our experimental results show that *1)* DAT is able to find a hyperoptimized configuration which produces shorter edit-scripts for 21.8% of the cases using the JDT meta-model (which is the meta-model used by default by GumTree). *2)* Local hyperparameter optimization is effective, as it allows finding shorter edit-scripts than the default configuration in up to ≈26% of cases.

To sum up, the contributions of this paper are:
- An original data-driven approach, called DAT for hyperparameter optimization in the context of AST differencing.
- An original and sound protocol for studying the performance of AST differencing, with cross-validation and statistical validation. The results of this experiment are available at https://github.com/martinezmatias/dat-experimental-results.
- A publicly available tool for hyper-optimization of AST differencing. It natively supports the popular GumTree AST differencing engine and provides extension points to integrate other differencing tools: https://github.com/martinezmatias/diff-auto-tuning.

The paper continues as follows. Section II presents the terminology used in the paper. Section III presents three cases in which state-of-the-art AST differencing produces incorrect results. Section IV discusses hyper-parameters of differencing algorithms. Section V presents DAT, our hyperparameter optimization approach for AST differencing. Section VI presents the results of the evaluation. Section VII presents the threats to validity. Section VIII presents the related work. Section IX concludes the paper.

II. TERMINOLOGY

In this section, we present the key terminology related to our contributions.

AST Differencing Algorithm

Computes the differences between two ASTs (Abstract Syntax Tree). For example, ChangeDistiller [21] and GumTree [17] are two popular AST diff algorithms. Typically, an AST diff algorithm has two input parameters, the two ASTs (AST_{left} and AST_{right}) generated from two source code files (F_{left} and F_{right}, respectively). Those ASTs are modeled using a *meta-model* (defined below). Finally, the AST algorithm outputs the differences between the two ASTs in the form of an *edit-script* (defined below). An AST differencing algorithm is implemented in a tool, for example, GumTree [17] is implemented in GumTreeDiff.[1]

[1] https://github.com/GumTreeDiff/gumtree

Edit-Script

Is the output of an AST diff algorithm. It represents the result of the comparison between the two ASTs given as input parameters. The edit-script is a sequence of edit operations applied to the AST nodes of AST_{left}. The operations are typically typed as {insert, remove, update, move}, and are meant to represent the transformation from AST_{left} into AST_{right}.

Matcher

For computing an edit-script between two ASTs, AST_{left} and AST_{right}, it is necessary to try to match (i.e., *link*) some tree nodes of AST_{left} with nodes from AST_{right}. The algorithm that executes this task is known as *matcher*. The criterion for matching two nodes depends on the matching algorithm, and can consider, for instance, the node's type and label, the topology, etc. The list of matched and unmatched nodes is then used for deducing an edit-script. For instance, those nodes from AST_{left} and AST_{right} that could not be matched correspond to removed or inserted nodes, respectively.

AST Meta-Model

Defines the structure of an AST. In particular, the meta-model describes: *a)* the possible types of AST nodes (e.g., invocations, assignments, methods); *b)* the attributes of each node type (e.g., a label); *c)* the optional and mandatory children of each node type, if any.

For example, a simplified meta-model for the Java language may have 4 node types for *classes*, *methods*, *fields* and *statements*; the *method* nodes have *a)* an attribute called *name*, *b)* a list of zero or more *statements* nodes as children.

Note that there can be several meta-models for representing ASTs of the same programming language. For example, GumTree can model a Java AST according to four different meta-models: JDT, Spoon, JavaParser and srcML. Since the choice of meta-model has an impact on the topology of the resulting ASTs, it finally also has an impact on the computed edit-scripts.

III. MOTIVATION

Previous work has revealed that state-of-the-art AST differencing algorithms generate inaccurate mappings, which impacts on the quality of the edit-scripts generated. For example, Fan et al. [18] show that GumTree [17] generates inaccurate mappings for 20%–29% of the file pairs analyzed.

In this Section, we present two cases for which GumTree with its default configuration produces incorrect or non-optimal edit-scripts. Then, in Section VI, we show how autotuning GumTree using our approach DAT allows GumTree to produce more understandable edit-scripts for these two cases.

A. Case 1: Spurious Add-Remove

Diff algorithms such as GumTree can produce edit-scripts with spurious edits [14]. A recurrent case of these edits is presented in Fig. 1, which shows the Interpreter.java file

TABLE I
HYPERPARAMETER SPACE FOR THE GUMTREE AST DIFFERENCING ALGORITHM

Hyperparameter	Description	Default	Values				
			Min	Max	Step		Total
Bottom-up Matcher	Bottom-up matcher used to compute the diff	Classic	{Classic, Simple, Hybrid}				3
STM_PC	Indicates the priority calculator used by the subtree matchers	Height	{Size, Height}				2
STM_MPTH	Threshold on the minimum priority value computed using STM_PC	1	1	5	1		5
BUM_SMT	Threshold on the minimum similarity between two AST nodes	0.5	0.1	1	0.1		10
BUM_SZT	Threshold on the maximum size of AST nodes to match	1000	100	2000	100		20

Fig. 1. (Case 1) Example of spurious add-remove edits found by GumTree using the default configuration and JDT meta-model. Other no-spurious edits are not presented in the figure.

Fig. 2. (Case 2) Visualization of the edits computed by GumTree using default configuration.

from the Log4J project. It presents the version from commit a04d92 (right part) and its previous version (left part). The figure also shows in a colored box the code affected by the edits obtained by GumTree after computing the diff of those two files, using the default configuration and the JDT metamodel. Each color represents a different edit type (red corresponds to remove, green to insert). The edit-script includes, in total, six edits. Three edits remove tree tokens (`public`, `class` and `Interpreter`), each one represented by an AST node (left part, in red), and the other three edits insert the same three tokens (left part, in green). All these are spurious edits. GumTree should not generate any of these six edits. As we discuss later in this paper, GumTree using the best configuration found by DAT does not produce those edits.

B. Case 2: Including Updates in the Edit-Script

We now focus on the differences that GumTree detects between the version of the file `PanelWindowContainer.java` from the jEdit project in commit 6867bd and its previous version.

The changes made by the developer were two: updates of the type of variables `dockables` and `buttons` from `ArrayList` to `List`. However, the diff produced by GumTree looks like that produced by a line-based diff such as GNU diff: The edit-script removes the fields from the left part, adds the fields from the right part (using the new field's type `List`), and moves the tokens related to the field names. This edit strip does not clearly express the changes done by the developers (i.e., update two nodes), and introduces spurious edits (e.g., the remove and add of tokens 'private').

C. Fixing Cases With Spurious Edits

AST differencing algorithms detecting move edits (such as GumTree) are all based on heuristics because this problem is NP-hard. The downside is that node mapping heuristics can behave badly and often make use of hyperparameters. In this paper, our goal is to optimize hyperparameters for improving the quality of the edit-scripts. Similar improvements may eventually also be achieved by adding new features to the algorithm or proposing new algorithms. Nevertheless, even the improved algorithms are likely to use hyperparameters, so even them can leverage our work on hyper-optimization.

IV. HYPER-PARAMETERS OF DIFFERENCING ALGORITHM

In this section, we describe the hyper-parameters of GumTree [17], one of the most used AST differencing algorithms. It has been used in hundreds of research works relying on AST differencing, according to the citations of the GumTree paper [17] collected by Google Scholar.

A. Hyperparameters

First, we specify the hyperparameter space of GumTree. This is done by analyzing the source code and discussing it with the lead developers of GumTree, one of whom is also coauthor of this paper. Table I lists and explains the GumTree hyperparameters. For each hyperparameter, the column Default shows the default value used in the implementation in version 2.1.2.[2]

1) Bottom-Up Matcher: The first hyperparameter of GumTree is the *bottom-up matching algorithm*. GumTree sequentially applies two types of matchers [17]:

1) Top-down matchers (also known as subtree matcher): find isomorphic subtrees of decreasing height or size. Mappings are established between the nodes of these isomorphic sub-trees. They are called *anchors* mappings.
2) Bottom-up matchers: navigate a tree in post-order (e.g., visit first leaves, then their parents, etc.) in order to match nodes not previously matched in the top-down phase. Two nodes match if their descendants (children of the nodes) include a large number of common anchors. Whenever a new mapping is established during this phase, a *recovery* phase is applied as the last chance to find mappings of the descendants of the nodes.

There is only one stable top-down matcher in GumTree, while there are three different stable bottom-up matchers:

---

[2]GumTree version considered: https://github.com/GumTreeDiff/gumtree/commit/ed3beeab1e00a31f23ab5e9a8292c3168221a1ca (July 2020).

classic, simple, and hybrid. These three matchers differ only in the way they apply the recovery phase.

2) Priority Calculator: During top-down matching, Gum-Tree greedily matches whole isomorphic subtrees. To establish the priority of the chosen subtrees, it uses a metric based upon the topology of the subtree: either its size (number of nodes in the subtree) or its height (length of the longest path from one leave to the root of the subtree). For example, when using size, GumTree will first try to find an isomorphic subtree for the subtrees with the largest number of nodes. This hyperparameter is called STM_PC.

3) Minimum Priority Threshold: As explained in the previous paragraph, GumTree uses a topological metric to order the subtrees to be matched by the top-down matcher. This hyperparameter indicates the minimum value of size or height (according to STM_PC) to be considered by the matcher. This hyperparameter is called STM_MPTH. The effect of the value depends on the chosen priority calculator. For example, GumTree configured with STM_PC = size and STM_MPTH = 3 will not consider subtrees with two nodes or less during top-down matching.

4) Minimum Similarity Threshold: A bottom-up matcher matches two AST nodes if *1)* they have the same type, and *2)* have a similarity greater than a threshold. Similarity is computed based on the common number of mapped descendants that both nodes have. Increasing this threshold implies that a bottom-up matcher increases the minimum ratio of common descendants and, consequently, tries to match more similar subtrees. The bottom-up matchers of GumTree obtain the similarity threshold in different ways. The greedy matcher uses the hyperparameter BUM_SMT to establish the similarity threshold. The default value is 0.5, which means that a bottom-up matcher only considers nodes that have, at least, a 50% of common descendants. The other two bottom-up matchers (Simple and Hybrid) automatically compute the threshold using the following formula: $threshold(t_1, t_2) = 1/(1 + log(desc(t_1) + desc(t_2)))$, where t_1, t_2 are subtrees, and $desc(t)$ gives the number of descendants of subtree t.

5) Maximum Size Threshold: As explained previously, once a bottom-up matcher finds, from a subtree on tree t_1, the most similar subtree from tree t_2, it applies a *recovery* phase (Section IV-1), which relies on an algorithm that searches for matches among the descendants of both subtrees that are still unmapped. Classic uses an optimal tree-edit distance algorithm which has a cubic complexity and, therefore, is slow on large subtrees. Simple uses an heuristic which is much faster than the optimal algorithm. Hybrid applies the algorithm of classic or simple, depending on the size of the subtree under consideration. Given the fact that classic and hybrid matchers can have a large running time if they try to apply the optimal tree-distance algorithm on large subtrees, they use the maximum size threshold hyperparameter, which establishes the maximum size of a subtree for which this algorithm is applied. This hyperparameter is called BUM_SZT and its default value is 1000.

B. Defining the Hyperparameter Domain

Table I shows the domain of each hyperparameter space. Some hyperparameters are numeric; in this case, we give the minimum and maximum values. In addition, we give a reasonable step value to explore the input domain for this parameter, this value was suggested and agreed on with the GumTree lead developer. For example, the hyperparameter BUM_SMT goes from 0.1 to 1, with steps 0.1, giving as a result the hyperparameters {0.1, 0.2, 0.3, ..., 0.9, 1}. For hyperparameters with categorical scale, we give a list of possible values. For example, the bottom-up matcher hyperparameter could receive three values: *Classic, Simple, or Hybrid*. The 'Total' column gives the number of values to explore per hyperparameter.

We recall that the Cartesian product on all hyperparameters creates all possible algorithm configurations that DAT evaluates. In total, we obtain 2210 different configurations in GumTree. Note that this total is not equivalent to the scalar multiplication of the values of each hyperparameter shown in the last column of Table I because there are dependencies between the hyperparameters. For example, the hyperparameter BUM_SZT is used by (*GreedyBottomUpMatcher* matcher but not by *SimpleBottomUpMatcher*. Thus, 2000 configurations correspond to ClassicGumTreeMatcher, 200 to HybridGumTreeMatcher and 10 to SimpleGumTreeMatcher,

C. Metamodels

In our experiments, we perform AST differencing on Java programs, as done in the original publication of GumTree [17]. GumTree supports multiple AST meta-models for Java code. The default one is called *JDT*, it is based on the Eclipse JDT Parser. The other meta-model we choose is the one defined using Spoon [45], an open-source library to analyze, rewrite, transform, and transpile Java source code.

V. DAT: AN APPROACH FOR HYPERPARAMETER OPTIMIZATION OF AST DIFFERENCING

We present DAT, an approach to optimize AST differencing algorithms which have configuration parameters such as [16], [17], [21], [24]. The main goal of DAT is to find the optimal algorithm configurations of an AST differencing algorithm in a data-driven manner. DAT finds the best algorithm configurations with respect to a benchmark of file pairs to diff.

A. AST Differencing Hyperparameters

An AST diff hyperparameter is a parameter whose value is used to control a particular procedure from an AST diff algorithm. A hyperparameter has a domain and a default value. The *default configuration* of a diff algorithm corresponds to the default values of the set of hyperparameters. Hyperparameters can be set using different mechanisms: *1)* at compile time, by writing the values directly in the code; *2)* at startup time, through a command-line option or environment variable; *3)* at run time, through a method call.

To better understand hyperparameters, we take the example of the GumTree algorithm. At some point, GumTree computes a mapping between nodes from the ASTs under comparison that have more than a certain ratio of common children. This threshold is called BUM_SMT, its domain goes from zero to one,
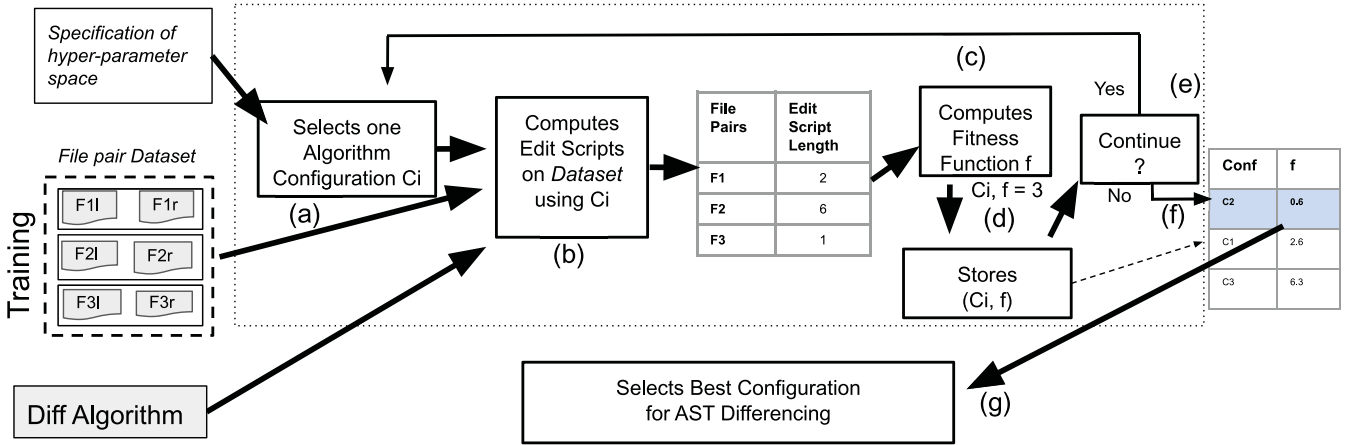
Fig. 3.   Workflow of DAT. It searches for the best AST differencing configuration in a data-driven manner, according to a set of file-pair.

and its default value is 0.5. This threshold is an hyperparameter of GumTree.

A **diff algorithm configuration** for an AST diff algorithm δ is a set of actual values for all available hyperparameters in δ. Hyperparameters are noted $h_1, h_2, \ldots, h_k$ and hyperparameter values are noted $v_1, v_2, \ldots, v_k$. A value v_i belongs to the **hyperparameter space** HS_i for h_i. The **hyperoptimization space** contains the Cartesian product of all hyperparameter spaces. The hyperoptimization space has n-dimensions, where each corresponds to a hyperparameter. A point in that space corresponds to a particular algorithm configuration of a diff algorithm. For each diff algorithm, there is one single point in its hyperparameter optimization space that corresponds to the *default* configuration.

An important dimension of AST differencing algorithms is the used *AST meta-model*. Two meta-models can produce, for the same source code file, two different ASTs in terms of topology (such as size or height). However, thresholds applied to values computed from the topological properties of the ASTs are very common in AST differencing algorithms. For instance, in GumTree at some point an optional mapping phase is launched depending on the number of children of the node under consideration (the BUM_SZT hyperparameter which default value is 1000). For instance, as the Spoon Java parser produces ASTs with more nodes than the JDT Java parser[3], it could affect the behavior of the algorithm. For this reason, we suspect that a first interesting case to apply hyperparameter optimization is at the level of each AST meta-model.

Another important dimension is the source code itself. For instance, a source code with very long methods will also induce a very different behavior of the algorithm, due to the previously mentioned threshold, compared to a source code with very short methods. Therefore, a second interesting case for hyperparameter optimization is at the level of given file pair for a given AST meta-model.

---

[3]We applied a statistical test to ensure that ASTs generated from the JDT meta-model (JDT) are different from those generated from the Spoon meta-model. As the distributions of AST' sizes and heights are not normalized, we use a Wilcoxon signed-rank test to reject (at 0.05 level) the two null-hypotheses which state that the size (resp. height) of a AST_{Spoon} is similar to the size (resp. height) of an AST_{JDT}.

We summarize these two use cases in the next section.

B. Use-Cases of DAT

There are two main use cases of DAT: *global* and *local* hyperparameter optimization.

For **global hyperparameter optimization**, the goal is to find a default configuration that is expected to work well for ASTs produced using a given meta-model. The global hyperparameter optimization is designed to assist AST differencing tools maintainers to compute good default values for the hyperparameters for a given AST meta-model. These values will then be automatically applied when using the diff tool to provide an improved end-user experience without suffering additional cost (the optimized values are computed on a training set and then stored in the diff tool).

For this use case DAT does a *set*-based optimization: it searches for the best-performing configuration over a set of training file pairs written in a same language and parsed using a same meta-model.

For **local hyperparameter optimization**, DAT does a *case*-based optimization: it searches for the best performing configuration on a given file pair, using a given meta-model. It is designed for end-users that want to compute the best possible diff at the expense of computation time (since in this setup the optimization is computed each time an end-user wants to diff a file pair).

C. Workflow

DAT takes as input: *1)* an *unlabelled* dataset of file pairs, *2)* a meta-model, *3)* an AST diff algorithm, and *4)* the specification of the algorithm's hyperparameter space. It outputs a sorted list of algorithm configurations, ordered in ascending order, from the best to the worst performing according to a given metric.

DAT implements the workflow presented in Fig. 3. Given a set of file pairs (i.e., a training dataset for data-driven optimization), and a specification of the hyperparameter space, DAT first selects one algorithm configuration, C_i, to evaluate (step a). The exploration and selection of other configurations is done according to a search technique (as described in Section V-E).

Then, DAT computes the edit-scripts for each file pair from the input dataset using the selected configuration C_i (step b). After having executed all the AST differencing tasks, DAT computes the $fitness$ value C_i, this fitness value then guides the search process. As the fitness value, by default DAT computes the averages of the length of those edit-scripts (step c), and stores that fitness value together with the configuration in a list (step d). Note that DAT can be extended to use another fitness function. Then, it may repeat the aforementioned step by selecting another algorithm configuration (step e), or it stops the search (step f). The stopping criterion depends on the search technique employed by DAT.

Finally, DAT sorts the list with the evaluated configurations according to their fitness values (i.e., the average edit-script lengths) in increasing order: algorithm configurations which produce shorter edit-scripts have better fitness than those algorithm configurations which produce larger edit-scripts on average.

The best algorithm configuration found by DAT in the previous phase (the first element from the returned list) can be used to test the optimization done by DAT on a separate dataset. For example, we can compare the performance on a testing dataset of the best configuration found (step g) with that one from the *default* configuration (step h). This best configuration can be used in production as the new *default* configuration of the AST diff algorithm under optimization.

In Sections V-D and V-E, we detail, respectively, the fitness function and the search techniques employed by DAT.

D. Fitness Function

The main challenge that AST diff hyperparameter optimization faces is the definition of a good fitness function. There is no ground truth on whether a hyperparameter configuration is good or not. DAT overcomes the lack of ground truth for hyperparameter optimization by using metrics that compare the outputs of two algorithm configurations (i.e., two edit-scripts). Based on the result of that comparison, DAT decides which of those algorithm configuration produces an edit-script of *higher quality*.

We use the *length of an edit-script* as the metric to guide hyperparameter optimization. As suggested by [17], the length of the edit-scripts is an indicator of the effort required to understand the changes between two files, because shorter edit-scripts are typically easier to understand. Moreover, it is an accepted proxy of the quality of the edit-scripts: other researchers have used it to measure the improvements introduced by new AST diff algorithms (e.g., [16], [23], [24], [25], [26], [38]).

DAT's fitness function takes three parameters: *i)* C: a single algorithm configuration (as defined in Section V-A). *ii)* TS: a training set composed of a set of file pairs. The diff algorithm, configured with C, is executed on each pair from TS. *iii)* M: a metric (e.g., average, median) that quantitatively measures the diff quality (i.e., the script length).

E. Search Techniques

DAT includes powerful search-based techniques used in hyperparameter optimization for other software engineering tasks (e.g. [3]): GridSearch, Hyperopt and Optuna. It focuses on techniques that are implemented in open-source generic off-the-shelf optimization frameworks.

1) Grid Search [8]*:* This optimization method exhaustively searches through a specified slice of the hyperparameter optimization space. Given the specification of a hyperparameter space, GridSearch first creates algorithm configurations by performing the Cartesian product between all the selected subsets of hyperparameters. For each algorithm configurations, it computes the fitness function described in Section V-D on a set of file-pairs (the training set). Finally, it returns the configurations with the highest fitness (the shortest edit script).

2) Hyperopt [10]*:* Is a distributed asynchronous hyperparameter optimization framework, based on the Tree-of-Parzen-Estimators (TPE) algorithm [8]. TPE is a Bayesian optimization approach [47], based on *Sequential Model-Based Optimization* [27]), which builds a probability model $p(score|hyperparameter)$ (aka the "surrogate" function) of the objective fitness function and uses it to select the most promising hyperparameters to evaluate in the objective function. The advantage of this algorithm is when evaluating expensive fitness functions. In these cases, w.r.t. Grid Search, it is less expensive because it explores a part of the search space, leading to lower execution costs [8]. As mentioned by [36] Hypeort is the most cited optimization algorithm.

3) Optuna [4]*:* Is an hyperparameter optimization framework designed by *define-by-run criteria* [4] (i.e., it allows the user to dynamically construct the search space [4]), and furnished with an efficient sampling algorithm and pruning algorithm. Optuna is also an extension of TPE.

We do not consider other optimization techniques recently presented, such as Dodge [1], niSneak [36], NUE[4] because, after inspecting their code, we found that they are not reusable and deeply tied to machine learning (ML) models from *scikit-learn* [46]. In this paper, we optimize diffs, not ML models. Thus, the utilization of techniques is not feasible without completely reimplementing them.

VI. EXPERIMENTAL RESULTS

In this section, we first present the data set we use for the evaluation of DAT. Then, we respond to the research questions.

A. Evaluation Dataset

The evaluation of DAT consists in running the hyperparameter optimization GumTree on a set of file-pairs. We create one as follows. First, we choose repositories of open-source projects in order to extract revisions of files. We choose CVSVintage [41], a dataset composed of 14 CVS repositories of open-source projects written in Java, because GumTree was initially evaluated on that dataset [17].

To create the set of file pairs, we first convert each CVS repository to a GIT repository using the tool cvs2git[5]. We were able to successfully convert 13 of the 14 repositories.

---

[4]Nue framework https://github.com/lyonva/Nue
[5]cvs2git: https://www.mcs.anl.gov/jacob/cvs2svn/cvs2git.html

Then, we navigate the history of each GIT repository, commit by commit, and for each one, we store the Java files that have been updated according to the command `git diff`. For each file f updated by commit C, we create a pair file (f_p, f), where f_p is the previous version of file f (i.e., the version before commit C). All file pairs are available in our appendix.

We study file-pairs that introduce AST changes. For detecting them, we compute the edit-script vanilla GumTree on each pair and keep those that have an edit-script longer than zero. Due to limited computational resources, we consider a subset of files. For that, we randomly select up to 1000 file-pairs per project. Nine projects have less than 1000, so we consider all of them for these projects. In total, we consider 9,052 file-pairs.

B. RQ1: To What Extent Does Hyperparameter Optimization Improve the Performance of AST Differencing?

1) Protocol for RQ1: To answer this research question, we execute the *global* hyperparameter optimization from DAT on the evaluation dataset described in Section VI-A, composed of 9,052 file-pairs. We performed this hyper-optimization for the two considered AST meta-models (JDT and Spoon) using the GridSearch technique implemented in DAT.

To minimize the risk of data overfitting, we apply a 10-fold cross-validation. For each fold, we first generate two sets: training (90% of the data) and testing (the remaining 10%). Then, we hyperoptimize GumTree using the training dataset with the goal of finding the configuration with the best performance C_{Best}. We compare the performance of configurations using two metrics: reduction of the *1)* median value (50th percentile) and *2)* 75th percentile value. Then, using the testing dataset, we calculate the performance of: *a)* the best configuration (C_{Best}), and *b)* the default configuration ($C_{Default}$). Next, we compute the proportion of file pairs where: *a)* hyper-optimization (C_{Best}) improves the edit-script w.r.t. default configuration ($C_{Default}$) (Metric I), *b)* hyper-optimization produces an equivalent edit-script (Metric E), *c)* hyper-optimization produces a worse edit-script (Metric W). Finally, we report the median of I, E and W over all folds.

We also proceed to a statistical assessment of the results using a Wilcoxon signed rank test against the size of the edit-scripts produced by two different configurations: the default and the best one found using the GridSearch technique. We use this test since we have no assumption about the distribution of the edit-script sizes. Our null and alternative hypotheses that we focus on in this RQ are as follows:

- H^1_{null} There is no difference between the median length of the edit-scripts produced using the global and default configuration (alternative H^1_{alt} the edit-scripts produced using global have a median shorter length than the ones produced using default).

We also report the effect size using Rosenthal's R, whose value varies from 0 (small effect) to close to 1 (large effect).

2) Results: Table II presents the results. It displays three columns that present the percentage of cases from the testing set where hyperoptimized GumTree finds: *a)* shorter edit-script than default GumTree, meaning that the

TABLE II
(RQ1) COMPARISON BETWEEN THE PERFORMANCE OF GLOBALLY HYPEROPTIMIZED GUMTREE AND DEFAULT GUMTREE. COLUMNS Med AND $75P$ ARE THE % OF IMPROVEMENT OF THE MEDIAN AND 75TH PERCENTILE VALUES, RESPECTIVELY

| Meta-Model | % Cases Where GridSearch Optimization | | | | | | p-value |
| | Improves (I) | | Equals (E) | | Worse (W) | | |
	Med	75^{th}P	Med	75^{th}P	Med	75^{th}P	
JDT	21.8	22.5	75.6	72.9	2.5	4.4	$2.2e-16$
Spoon	16.1	16.3	81.8	81.4	2.2	2.1	$2.2e-16$

TABLE III
(RQ1) GLOBAL CONFIGURATIONS FOUND BY DAT USING GRIDSEARCH FOR JDT AND SPOON JAVA META-MODELS. SYMBOL '-' MEANS THAT THE CONFIGURATION DOES NOT USE THE HYPER-PARAMETER

Hyperparameter	Default	Best for JDT	Best for Spoon
Matching Algorithm	Classic	Hybrid	Classic
STM_ PC	Height	Size	Size
STM MPTH	1	1	1
BUM_SMT	0.5	-	0.2
BUM_SZT	1000	400	600

hyperoptimization improves the performance of GumTree (column I), *b)* the same length edit-script (column E), *c)* a larger edit-script, meaning that the hyperoptimization harms the default configuration of the differencing algorithm (column W). The table reports two metrics: improvement on the *1)* median, *2)* 75th percentile value of edit-script length. For a matter of space, we discuss the former metric.

To diff ASTs from JDT, the optimization of GumTree improves the performance of the default GumTree for 21.8% of cases. The detriment of applying global optimization is much lower: in only 2.5% of the cases, hyperoptimized GumTree produces larger edit-scripts. For the rest of the cases (75.6%), hyperoptimization has no impact on the length of edit-script produced by GumTree. Table III shows the best configurations found by DAT for the JDT and Spoon meta-models, and the default value used by GumTree. We observe that for JDT the best configuration uses a different matching algorithm (Hybrid) than the default configuration (Classic).

We use a Wilcoxon signed rank test to statistically assess the differences in edit-script lengths produced using the hyperoptimized configuration versus the default configuration. The obtained P-value is $2.2e-16$, therefore, we reject the null hypothesis H^1_{null}. The effect size, calculated using Rosenthal's R, is -0.585, which can be considered between medium and large.

To diff ASTs designed with the Spoon meta-model, hyperoptimization with DAT has less impact on the number of improved cases (16.1%). It means that the default configuration of GumTree works already well in most cases. We observe from Table III that the best configuration for Spoon has the same matching algorithm as the default (Classic). However, there are three parameters that receive different values: STM_ PC, BUM_SMT, and BUM_SZT.

Again, we run a Wilcoxon signed rank test on the edit-script length distribution and the obtained P-value is inferior to
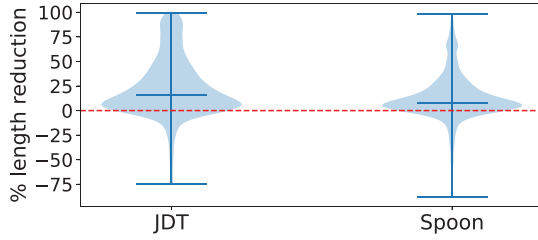
Fig. 4. Distribution of the percentage of reduction of the edit-script length using the optimized configuration w.r.t. the default configuration. (Cases with no improvement or detriment are ignored).
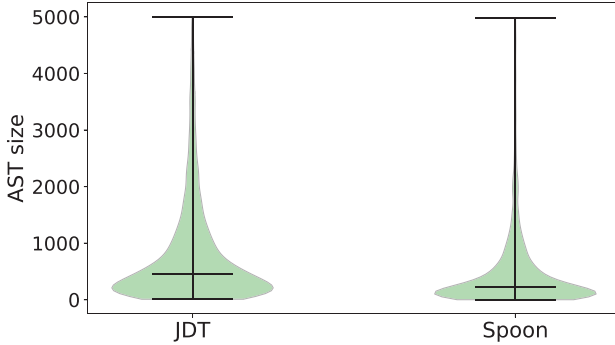

Fig. 5. Distribution of AST sizes (# of nodes).

$2.2e - 16$, therefore, we reject the null hypothesis H^1_{null}. The effect size, computed using Rosenthal's R is -0.669, which can be considered between medium and large.

Fig. 4 shows the distribution of the percentage of reduction in the size of the edit-scripts calculated with the optimized configuration compared to the edit-script calculated with the default configuration. The figure considers the cases that present improvement due to the optimization process (these have % positive and correspond to 21.8% in JDT and 16.1% in Spoon), cases for which optimization produces worse results (% negative, 2.5% in JDT and 2.2% in Spoon), but ignores those with equal results to facilitate visualization. For JDT, half of the optimization-affected cases have a reduction of at least 20%, and there is a considerable number of cases with an improvement greater than 75%. In contrast, for Spoon, the reduction in the size of the edit-scripts is less than for JDT, and there are fewer cases that are reduced by more than 50%. That can be explained by the fact that Spoon ASTs are smaller. Fig. 5 shows the distribution of the AST sizes (expressed in # of nodes). The JDT ASTs have a median of 480 nodes, while Spoon ASTs have 235. Consequently, a change on Spoon ASTs typically spans a smaller number of nodes than on JDT.

We also inspected cases in which the tuning resulted in longer edit scripts. We observe that this happens when a file-pair under analysis has an insertion of code that already exists in the file. The tuned version of GumTree creates spurious moves and additional inserts related to this code.

In conclusion, this experiment shows that the effect of the optimization is different according to the meta-model used: for JDT, the optimization a) affected more cases, and b) for those affected cases, the reduction is more significant.

Method	Evals /Diffs	JDT		Spoon	
		Med	75thP	Med	75thP
GridSearch	2210	21.9	22.5	15.9	16.24
Hyperopt	100	20.2	22.8	15.2	16.13
Optuna	100	20.2	22.8	15.8	16.07

Answer to RQ1: Global hyperparameter optimization improves GumTree performance by providing a better configuration than the default configuration. The hyper-optimized configuration produces the shortest edit-scripts for 21.8% of the cases using the JDT meta-model and for 16.1% of the cases based on using the Spoon meta-model, in a statistically significant manner.

Implications for practitioners: Maintainers and users of AST differencing tools such as GumTree have a new tool in their toolbox. When they apply AST differencing to a new programming language, they can first perform a global hyperparameter optimization, which would identify a new configuration that is *1)* better than the default, and *2)* tuned to a given AST meta-model.

C. RQ2: To What Extent Can One Speed up Hyperoptimization Using More Advanced Techniques Than Exhaustive Grid Search?

1) Protocol for RQ2: In this research question, we evaluate two other hyper-parameter optimization techniques provided by DAT: Hyperopt [10] and Optuna [4]. Rather than evaluating all algorithm configurations as GridSearch does (in total, 2210 as described in Section IV-2), these techniques are able to perform fewer evaluations. This number is given to Hyperopt and Optuna as input parameters. In this experiment, we set up 100 evaluations, which corresponds to the default value from Hyperopt. Doing 100 evals per diff means that Hyperopt and Optuna will do just the 4.5% of the evaluations that GridSearch performs per diff.

2) Results: Table IV shows the percentage of improved cases of GumTree using the best configuration found by Grid-Search, Hyperopts and Optima compared to the default configuration of GumTree. We present the results from diffs computed on JDT and Spoon ASTs. The column *Eval/diffs* shows the number of diffs executed by each technique in each file-pair from the training dataset. We recall that each execution has a different algorithm configuration. The column *Med* means the percentage of cases from the testing set where the best configuration reduces the median value of the fitness function

(in our case, the length of the edit-script), and the column *75thP* the percentage of cases where the best configuration reduces the 75th percentile value of the fitness function.

First of all, we observe that the values of the two metrics using any of the optimization techniques are similar, for the two meta-models. For JDT, the median value obtained using GridSearch is slightly higher than the value obtained using Hyperopt and Optima: 21.9% and 20.2%. However, the number of executions per diff that the latter requires is much lower: 100 vs. 2210. This means that, by using Hypeorpt or Optuna, one can optimize GumTree with much less computational effort. For JDT, we do not observe any significant differences between Hyperopt and Optuna on the two metrics.

For Spoon, we observe that: *a)* the percentage of improvement is lower than that one obtained for JDT ($\approx$20% vs $\approx$15%), *b)* Optuna has almost the same rate of improvement as GridSearch for the median metric (15.9% vs 15.8%), *c)* Optuna reports a higher median value (15.8% vs 15.2%), but Hyperopt reports a higher improvement on the 75th percentile (16.13% vs 16.07%).

> **Answer to RQ2:** Using Hyperopt and Optuna, DAT can significantly reduce the number of evaluations performed to find the best configuration of GumTree, resulting in faster hyperoptimization.

> **Implications for practitioners:** To perform a global hyperparameter optimization for a new programming language or meta-model, it is enough to collect less than 10000 file-pairs (diffs) and use Hyperopt or Optuna. This optimization is done only once for a given programming language and meta-model.

D. RQ3: To What Extent Is Local Hyperparameter Optimization Effective?

1) Protocol for RQ3: To answer this research question, we extend the protocol used to answer RQ1 (Section VI.B.1) to find the best local configuration for each file pair from the training set. As before, we use GridSearch, Hyperopt and Optuna as optimization techniques. In particular, for each of those file pair $P = (f_p, f)$, we compare the fitness value (i.e., length of the edit-script) given by the local search on P with the value obtained using the default configuration on P.

We also proceed to a statistical assessment of the results using a Wilcoxon signed rank test against the size of the edit-scripts produced by the different configurations. Our null and alternative hypotheses that are studied in this research question are:

- H_{null}^2: There is no difference between the median length of the edit-scripts produced using the local and the default configuration (alternative H_{alt}^2 the edit-scripts produced using length have a median shorter length than those produced using the default).

2) Results: Table V presents the results of this experiment. It contains two sets of columns that present, for both the JDT

TABLE V
(RQ3) PERCENTAGES OF CASES IMPROVED BY APPLYING LOCAL OPTIMIZATION WITH DAT VS. DEFAULT CONFIGURATION

Comparison	% Improved (I)		% Equal (E)	
	JDT	Spoon	JDT	Spoon
DAT Local GridSearch vs Default	27.4	18.7	72.5	81.3
DAT Local Hyperopt vs Default	26.1	16.7	73.8	83.3
DAT Local Optuna vs Default	26.1	17.3	73.8	82.6

TABLE VI
(RQ3) MOST FREQUENT LOCAL HYPEROPTIMIZED CONFIGURATIONS FOUND BY DAT FOR THE JDT AND SPOON META-MODELS

Hyperparameter	Default	JDT	Spoon
Matching Algorithm	Classic	Classic	Classic
STM PC	Height	Size	Size
STM MPTH	1	1	1
BUM_SMT	0.5	0.1	0.1
BUM_SZT	1000	1100	1000

and Spoon meta-models, the percentages of file-pairs for which hyperoptimization improves the results (I) or produces equal results (E). We remark that local optimization from DAT never worsens the performance of AST differencing. This is by construction, as it optimizes a single file-pair, if the technique does not find a configuration that improves the default, then the default is used.

The row DAT **Local GridSearch vs. Default** presents the results obtained from local optimization. Local optimization positively impacts the performance of GumTree, producing shorter edit-scripts than those from the default configuration for 27.4% of cases with JDT and for 18.7% of cases with Spoon. We note that the improvement in local hyperoptimization is larger than that provided by global hyperoptimization (27.4 $\gg$ 21.8%). Using the Wilcoxon signed rank test, we reject the null hypothesis H_{null}^2 for both the JDT and the SPOON meta-models. The effect sizes computed using Rosenthal's R are –0.579 and –0.668 respectively, which can be considered between medium and large.

The last two rows in Table V present the results of local optimization using Hyperopt and Optuna. First, we observe that using these techniques, DAT is still able to locally improve the edit-script, but the percentage of improved cases is a bit lower than those using GridSearch achieve almost the same improvement that using GridSearch (e.g., for JDT, Hyperopt achieves 26.1% vs 27.4%). However, the number of evaluations (i.e., executions of diff) per file-pair under comparison is much lower using Hyperopt or Optuna: 100 for Hyperopt-TPE (default number of evaluations) vs. 2210 for GridSearch (we recall that it does an exhaustive search on the configuration space). Second, we observe that Optuna achieves slightly better results than Hyperopt on Spoon, and similar on JDT. We have observed this trend in global optimization (Section VI-C).

Developers and practitioners can decide the search method used by DAT according to the scenario they apply DAT, for example, a scenario with a limited budget or where they need to

TABLE VII
(RQ4) Execution Times of DAT Using Hyperopt and Optuna in Seconds (Columns t). The Columns 'Diff' Show the Median Time for Executing a Single Diff. Columns I Show the Percentage of Cases Improved Using Local Optimization w.r.t. the Default Configuration

# Evals	JDT					Spoon				
	Diff	Hyperopt		Opt		Diff	Hyperopt		Opt	
	t	t	I	t	I	t	t	I	t	I
100		9.85s	27	8.71s	27		4.01s	15	3.58s	15
50	0.065s	6.5s	27	5.94s	27	0.027s	2.97s	15	2.8s	14
25		2.8s	27	2.24s	27		1.54s	12	1.19s	14
10		1.39s	26	1.02s	23		1.05s	12	0.89s	9

optimize fast, thus techniques such as Hyperopt or Optuna, both based on TPE, would be more convenient, or another where they need to obtain the best result and do not have budget restrictions, thus GridSearch would be a better option.

Answer to RQ3: Local hyperparameter optimization for a single differencing task is doable: it allows practitioners to find shorter edit-scripts than the default configuration 26.1-27.4% of cases for the JDT meta-model and 16.7-18.7% for the Spoon meta-model.

Implications for practitioners: In sensitive downstream tasks, we strongly advise using local hyperparameter optimization for AST differencing in order to obtain the best edit-script according to a fitness function specific to the downstream task.

E. RQ4: What Is the Overhead of Local Hyperparameter Optimization?

1) Protocol: We focus on HyperOpt and Optuna because, as shown previously in RQ1, they achieve similar performance compared to GridSearch but execute fewer evaluations. We start running them using the default number of evaluations (100 evals). We compute, for each file-pair, the execution time of hyper-optimizing and the execution time of computing a diff using vanilla GumTree (i.e., using the default hyper-parameters) and then report the median values. Then, we measure the impact of reducing the number of evaluations on execution time and % of improvements. We run both HyperOpt and Optuna using 10, 25 and 50 evals per file-pair. Due to the large magnitude of our experiment (2 tools X 4 # of evals), we execute the whole experiment on 100 file-pairs randomly selected from our dataset.

2) Results: Table VII shows the median execution times (in seconds) of: *a)* executing one diff, *b)* execution local optimization using DAT Hyperopt and Optuna.

We first focus on row "#evals 100", which is the default value. We observe that the median diff time takes a few milliseconds (0.065 and 0.027 seconds for JDT and Spoon, respectively). Running hyper-optimization using 100 evals requires less than 10 seconds. This includes the time for: *1)* determining the

configurations to execute on each step, and *2)* executing 100 diffs using these configurations. We also observe that hyper-optimizing the Spoon metamodel is faster than optimizing JDT. The reason is that executing a diff on a Spoon AST is faster than on its corresponding JDT AST. This happens because, as shown in Fig. 5, Spoon ASTs are smaller than JDT ASTs.

As this optimization time could be too long in practice, we try lower numbers of evaluations per diff: 10, 25 and 50. Table VII shows the median time to optimize the diff using these numbers of evaluations (columns t) and the number of diffs improved by local optimization (columns I). We find that using less evaluations, in particular 25, both Hyperopt and Optuna: *a)* keep the acceptable performance w.r.t 100 evaluations (in JDT, DAT has the same % of improvement, while for Spoon this percentage is slightly reduced), and *b)* reduce the optimization time of a diff to less than 2 and 3 seconds for Spoon and JDT, respectively.

Furthermore, using 10 evaluations, local optimization is capable of working, in a median, below 1.5 seconds for HyperOpt and below ≈1 second for Optuna and is still capable of producing improvements on diffs. The reduction in the number of evaluations impacts more on Optuna than on Hyperopt. For Hyperopt in JDT, there is only one case (out of the 27 improved using 25 evals) that is not improved. In contrast, Optuna cannot improve 4 cases (out of the 27 improved using 25 evals) on JDT and 5 cases (out of the 14 improved using 25 evals).

These setups (10 and 25 evals) make DAT usable in practice, in our opinion. As the number of evaluations is a hyperparameter of DAT local optimization, the user can adjust it according to his/her requirements: either they want much better diffs or fast results.

Answer to RQ4: Local hyperparameter optimization, using Optuna with 25 evaluations, takes a median overhead of 1.19 and 2.24 seconds per diff for Spoon and JDT meta-models, respectively. Moreover, DAT using Optuna for local optimization is faster than using Hyperopt.

Implications for practitioners: In tasks such as visual representations of an AST diff (e.g., using the GumTree CLI), local optimization provides developers with an optimized AST diff within a few seconds.

F. Discussion About Trade-Off Between Local and Global Optimization

We consider that global and local optimization can be used in a complementary way and none replaces the other. Global optimization is designed to be executed when a new language or metamodel is introduced to an AST differencing algorithm, with the goal of learning the new default configuration for that novel language. Also, global optimization can be used when an AST diff algorithm is deployed in the infrastructure of an organization. There, global optimization is executed on existing code (e.g., code repositories) from that organization in order to learn its peculiarities. For example, limiting the length of methods in order to avoid the code smell 'long method' will impact on the size and length of the ASTs from that code. Hyperoptimizing the parameters that consider these variables (e.g. BUM_SZT) would potentially lead to an improvement in the AST diffs.

In contrast, we consider that an appropriate use case for local optimization is when a developer detects anomalies on a diff (e.g., a corner case, potentially generated using the global configuration) and wants to improve it. There, the local optimization is executed *on demand* and it is fast enough. To that extent, local optimization is not meant to replace global optimization but to complement it.

G. Analysis of the Cases Studies

In this section, we discuss how the best global hyperparameter configuration found by DAT helps GumTree produce a different output from the default configuration in each of the cases presented in Section III.

1) Case 1: Spurious Add-Remove: As described in Section III-A, GumTree, using both JDT and Spoon metamodels, produces six spurious edits that must not be produced. GumTree using the best configuration found by DAT does not produce them. The reason for having different edit-scripts when GumTree uses default and the best configuration is the following.

The three AST nodes in Section III-A cannot be mapped during the first phase (top-down matching explained in Section IV-1). Both Classic (default) and Hybrid (the matcher used by the best configuration found by DAT) matchers apply the same top-down matching strategy: Greedy subtree matching. In the second matching phase (bottom-up), the Hybrid matcher is able to map those nodes during *recovery phase* (a last step done by a matcher which tries to map the unmapped children of two subtrees whose roots are mapped). However, the Classic matcher does not map them because it invokes *recovery phase* only if the size of the trees to match is smaller than the hyperparameter BUM_SZT (by default 1000). The size of the parent tree of these three mentioned nodes (which corresponds to the class Interpreter) is greater than BUM_SZT=1000, so the *recovery phase* is never called and the three nodes remain unmapped.

2) Case 2: Including Updates in the Edit-Script: The edit-script generated by GumTree using the best configuration found in Section VI is much shorter and understandable: as Fig. 6

Fig. 6. (Case 2) Visualization of edits by GumTree (two updates) between the file PanelWindowContainer.java (jEdit project) from commit 6867bd (right) and its previous version (left) using the best global configuration.

shows, it only includes one update of each field declaration statement rather than adding and removing operating.

During the top-down matching phase using the default configuration, GumTree maps two nodes (one from the left tree, the other from the right tree) if their labels are equal and, following the default values of hyperparameters STM_PC and STM_MPTH, their heights are >= 1. Note that using this configuration, all leaf nodes (even those that are not modified) from the left tree are not mapped to any from the right tree because their heights are 0. Then, during the bottom-up matching match, GumTree cannot match any of the nodes corresponding to the field declaration statements private ArrayList dockables; and private ArrayList buttons; from the left part with those corresponding to the field declarations from the right (private List dockables; and private List buttons;). This is because the similarity value between the field declarations from the left and right (e.g., private ArrayList dockables; and private List dockables;, respectively) is 0.4, which is lower than the default threshold BUM_SMT (i.e. 0.5), which controls the mapping of two nodes. This low similarity value is due to the fact that most of the descendants have not been mapped. As a result, the edit-script generated using the algorithm of Chawathe et al. [11] (Fig. 2) includes remove and add edits that affect these unmapped nodes, including the field declaration.

GumTree tuned with the best configuration, which uses the Hybrid Bottom-up matcher, finds the expected edit-script (Fig. 6). When the tuned version of GumTree uses 'size' = 1, it arrives to match leaves nodes that are not mapped using the default configuration ('height' =1).The map is possible because the size of a leaf node is 1. These mappings produced by GumTree with the tuned version of GumTree but not with the default impact bottom-up matching: the similarity score between the declaration statements (e.g., private ArrayList dockables; and private List dockables;) is 0.6, higher than the threshold BUM_SMT with value 0.5. For this reason, both statements are mapped, and the edit-script generator does not generate spurious add and remove edits.

VII. THREATS TO VALIDITY

A. Construct Validity

Fitness function: The fitness function is meant to measure the quality of an edit-script. We decided to use the length of the editing scripts as a fitness function because it has previously been used in the literature for this purpose [16], [17], [23],

[24], [25], [26], [38]. Other fitness functions might yield different outcomes.

B. Internal Validity

Hyperoptimization techniques: DAT provides optimization using GridSearch, Hyperopts, and Optuna, which have been proven to be successful in hyperparameter optimization for software engineering tasks [35], [36], [50]. There may exist other search techniques that produce better results. Recall that GridSearch, Hyperopts, and Optuna are the only generic, off-the-shelf implementations publicly available. Others, including Dodge [1] or niSNEAK [36], provide implementation tied to machine learning frameworks such as scikit-learn, which makes them unusable for optimizing AST differencing.

Quantization of the hyperparameter space: We quantize the hyperparameter space of GumTree using *initial*, *end* and *step* values (all presented Table I). It could be the case that there are values not included in the selected subset that produce better values. As Bergstra [9] remarks, zones of useful hyperparameters might be very narrow within the full search space. Consequently, considering large steps may not help DAT to find such zones. To test whether that happens, we replicate our experiment from RQ4, this time running Hyperopt using a more refined hyperparameter space: we change the steps of hyperparameter BUM_SMT from 0.1 to 0.01 and of BUM_SZT from 100 to 10. Using Hyperopt with 25 and 100 evals per diff, we observe that the refinement does not improve the results: It either *a)* maintains the same performance, or *b)* reduces the performance up to 2%. This happens because the refinement enlarged the search space.

C. External Validity

AST metamodeling: Our results show the importance of AST meta-modeling in differencing. We selected two meta-models for modeling Java ASTs. We are aware that there exist other meta-models (e.g., JavaParser). As shown by the clear performance difference between JDT and Spoon, it may happen that hyperparameter optimization by DAT performs differently for other Java meta-models.

Selection of Java: Our results are made on AST differencing for Java programs, as in most related work on AST differencing analysis [17], [21], [23], [26]. Future experiments will improve the external validity in other programming languages.

Evaluation dataset: We evaluate DAT on a dataset composed of 14 popular open-source projects, because GumTree, the algorithm that we hyper-optimize, was initially evaluated on the same dataset. Future experiments will improve the external validity in other datasets.

VIII. RELATED WORK

A. Advanced AST Differencing Algorithms

Since the preliminary work by Chawathe et al. [12], two AST differencing algorithms with a large impact recently are GumTree [17] and ChangeDistiller [21]. Several works have extended ChangeDistiller and GumTree with the goal of improving their performance. For example, Higo et al. [25] extend GumTree by adding a new edit type: copy-and-paste. They found that 18% of the edit-scripts generated using their approach are shorter than those from GumTree. Dotzler et al. [16] present MTDIFF, based on ChangeDistiller, which improves the detection of moved code and is able to reduce the length of edit-scripts. Matsumoto et al. [38] present a hybrid AST diff approach that matches the AST nodes with information from the *diff* command (based on the Myers algorithm [42]), and is able to generate shorter edit-scripts than GumTree for the 30–50% of the cases analyzed. Yang and Whitehead [53] use textual-differencing to prune the AST. This reduces both the number of nodes and the execution time.

Frick et al. [23] present an extension of GumTree, Iterative Java Matcher (IJM), which produces more accurate and compact edit-scripts by optimizing move and update edits. Huang et al. present ClDiff [26], an AST differencing algorithm that produces concise edit-scripts by grouping and linking AST nodes affected by changes. It generates shorter edit-scripts than GumTree for 48% file-pairs analyzed.

De la Torre et al. [14] study the quality edit-script generated by GumTree and propose four categories of imprecisions on edit-scripts. They empirically study the presence of such imprecision in C# programs. Fan et al. [18] define an approach that calculates statements with inaccurate mappings for AST differencing algorithms. They found that GumTree generates inaccurate mappings for 20%–29% of the cases.

We have the same goal as most of these papers: reducing the length of the edit-scripts. Yet, even if some of this related work does manual tweaking of hyperparameters, none of those papers does automated hyperoptimization. Our experiment in this paper shows that the default configuration is indeed suboptimal. We note that our contribution to AST differencing hyperoptimization is applicable to all the past and future AST differencing algorithms to come.

B. Hyperparameter Optimization in Machine Learning and Software Engineering

Hyperparameter optimization has been applied in various areas, notably in machine learning. For example, Kotthoff et al. [31] presented Auto-Weka, a tool that automatically applies hyperparameter optimization to machine learning models from Weka. Feurer et al. [19] applies hyperparameter optimization to models of the Sklearn framework.

Now, let us focus on hyperparameter optimization in software engineering. Tantithamthavorn et al. [50] studied the impact of automated parameter optimization on defect prediction models. Their study shows that automated parameter optimization can have a large impact on performance stability and interpretation of defect prediction models. Similarly, Li et al. [35] studied show that parameter optimization improves the performance of cross-project defect prediction techniques.

Arcuri and Fraser [6] applied hyperparameter optimization on EvoSuite [22] a framework for test generation. They showed

that this positively impacts the performance of EvoSuite. However, the author, as also found by Kotelyanskii and Kapfhammer [30], stressed that the parameter settings obtained may be worse than arbitrary default values. This has motivated us to precisely measure those cases where hyperoptimization produces worse results than the default configuration of GumTree. Also related to software testing, Jia et al. [28] used hyperparameter optimization for Combinatorial Interaction Testing (CIT).

The width of the software engineering domains where hyperoptimization has been used is large. Apel et al. [5] integrated hyperparameter optimization in JDime, a tool for structured merge. Shu et al. [48] applied hyperparameter optimization to improve data preprocessing for software bug report classification. Basios et al. [7] applied optimization to the problem of selecting data structures that share a common interface. Panichella [44] carried out hyper-optimization of LDA (Latent Dirichlet Allocation) applied to identify duplicate bug reports. [52] applied 12 hyperparameter optimization techniques to the software effort estimation task.

Other works have presented hyperparameter optimization techniques that were evaluated in software engineering tasks. Agrawal et al. [2] presented SMOTUNED, an approach that tunes SMOTE [13], an oversampling technique to fix data imbalance, and evaluated on the defect prediction task. Then, Agrawal et al. [1] presented Dodge, hyperparameter optimization for machine learning. Dodge detects and ignores redundant *tunings* (i.e. configurations that lead to indistinguishable results) and runs orders of magnitude faster without harming the performance of the approaches under tuning. Dodge was evaluated in software defect prediction and text mining [1], later in bad smell detection, predicting GitHub issue close time, bug report analysis, and defect prediction [3]. Yedida and Menzies [54] presented GHOST, a method that relies on a combination of hyper-parameter optimization of feedforward neural networks and a novel oversampling technique. Lustosa and Menzies [36] present the hyperparameter optimization niSNEAK, based on landscape analytics, which was evaluated on the task of predicting the health of open source projects. The implementations of these tools are tied to the optimization of machine learning models (e.g., Scikit-learn [46] models). This prevents us from applying them in AST differencing.

To our knowledge, we are the first to propose and comprehensively study on hyperparameter optimization for AST differencing.

IX. CONCLUSION

In this paper, we have proposed to use hyperoptimization for AST differencing. We have described a novel approach, called `DAT`, consisting of *1)* specifying the hyperparameter space of an AST differencing algorithm, *2)* applying a search technique to hyperoptimize the algorithm in a data-driven way based on a training dataset of AST differencing cases. The approach has been instantiated for the popular AST differencing algorithm GumTree [17]. We have performed a comprehensive quantitative assessment of `DAT`, which shows that hyperoptimization

improves the AST edit-scripts in up to 21.8% of the differencing tasks using global hyperoptimization and up to 27% using local hyperoptimization. Our technique is widely applicable to all AST differencing algorithms: it can benefit both already proposed AST differencing systems and future ones to come. The main direction for future work is to apply and evaluate hyperparameter optimization of AST differencing on other real-world scenarios, such as the improvement of downstream tasks using hyperoptimized AST edit-scripts (e.g., automated repair, bug fix analysis).

REFERENCES

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to 'dodge' complex software analytics," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2182–2194, Oct. 2021.

[2] A. Agrawal and T. Menzies, "Is 'better data' better than 'better data miners'? On the benefits of tuning smote for defect prediction," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2018, pp. 1050–1061. [Online]. Available: https://doi.org/10.1145/3180155.3180197

[3] A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, and T. Menzies, "Simpler hyperparameter optimization for software analytics: Why, how, when?," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2939–2954, Aug. 2022.

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Min. (KDD)*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 2623–2631.

[5] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. New York, NY, USA: ACM, 2012, pp. 120–129. [Online]. Available: https://doi.org/10.1145/2351676.2351694

[6] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Softw. Eng.*, vol. 18, pp. 594–623, 2013. [Online]. Available: https://doi.org/10.1007/s10664-013-9249-9

[7] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, "Darwinian data structure selection," in *Proc. 26th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: ACM, 2018, pp. 118–128. [Online]. Available: https://doi.org/10.1145/3236024.3236043

[8] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proc. 24th Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates, Inc., 2011, pp. 2546–2554.

[9] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[10] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. Cox, "Hyperopt: A python library for model selection and hyperparameter optimization," *Comput. Sci. Discovery*, vol. 8, Jul. 2015, Art. no. 014008.

[11] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD Rec.*, vol. 25, no. 2, pp. 493–504, Jun. 1996. [Online]. Available: https://doi.org/10.1145/235968.233366

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 1996, pp. 493–504. [Online]. Available: https://doi.org/10.1145/233269.233366

[13] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, no. 1, pp. 321–357, Jun. 2002.

[14] G. de la Torre, R. Robbes, and A. Bergel, "Imprecisions diagnostic in source code deltas," in *Proc. Int. Conf. Min. Softw. Repositories (MSR)*. New York, NY, USA: ACM, 2018, pp. 492–502. [Online]. Available: https://doi.org/10.1145/3196398.3196404

[15] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcDiff: A syntactic differencing approach to improve the understandability of deltas," *J. Softw.: Evol. Process*, vol. 32, no. 4, 2020, Art. no. e2226.

[16] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. Piscataway, NJ, USA: IEEE, 2016, pp. 660–671.

[17] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 313–324.

[18] Y. Fan, X. Xia, D. Lo, A. E. Hassan, Y. Wang, and S. Li, "A differential testing approach for evaluating abstract syntax tree mapping algorithms," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 1174–1185.

[19] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 2, Cambridge, MA, USA: MIT Press, 2015, pp. 2755–2763.

[20] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *Proc. Work. Conf. Reverse Eng. (WCRE)*, 2007, pp. 70–79.

[21] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[22] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[23] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *Proc. Int. Conf. Softw. Maintenance Evol. (ICSME)*. Piscataway, NJ, USA: IEEE, 2018, pp. 264–274.

[24] M. Hashimoto and A. Mori, "Diff/TS: A tool for fine-grained structural change analysis," in *Proc. 15th Working Conf. Reverse Eng.* Piscataway, NJ, USA: IEEE, 2008, pp. 279–288.

[25] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler AST edit scripts by considering copy-and-paste," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*. Piscataway, NJ, USA: IEEE, 2017, pp. 532–542.

[26] K. Huang et al., "ClDiff: Generating concise linked code differences," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 679–690.

[27] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. 5th Int. Conf. Learn. Intell. Optim. (LION)*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 507–523. [Online]. Available: https://doi.org/10.1007/978-3-642-25566-3_40

[28] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 540–550.

[29] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, 2011, pp. 151–160.

[30] A. Kotelyanskii and G. M. Kapfhammer, "Parameter tuning for search-based test-data generation revisited: Support for previous results," in *Proc. 14th Int. Conf. Qual. Softw.*, 2014, pp. 79–84.

[31] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA," *J. Mach. Learn. Res.*, vol. 18, no. 25, pp. 826–830, Jan. 2017. [Online]. Available: http://jmlr.org/papers/v18/16-261.html

[32] A. Koyuncu et al., "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Softw. Eng. J.*, vol. 25, no. 3, pp. 1980–2024, May 2020. [Online]. Available: http://arxiv.org/pdf/1810.01791

[33] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proc. 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, 2016, pp. 213–224.

[34] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proc. 11th Joint Meet. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: ACM, 2017, pp. 593–604. [Online]. Available: https://doi.org/10.1145/3106237.3106309

[35] K. Li, Z. Xiang, T. Chen, S. Wang, and K. C. Tan, "Understanding the automated parameter optimization on transfer learning for CPDP: An empirical study," 2020, *arXiv:abs/2002.03148*. https://arxiv.org/abs/2002.03148

[36] A. Lustosa and T. Menzies, "Optimizing predictions for very small data sets: A case study on open-source project health prediction," 2023, *arXiv:2301.06577*. https://arxiv.org/abs/2301.06577

[37] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 176–205, Feb. 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9282-8

[38] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond GumTree: A hybrid approach to generate edit scripts," in *Proc. IEEE/ACM 16th Int. Conf. Min. Softw. Repositories (MSR)*. Piscataway, NJ, USA: IEEE, 2019, pp. 550–554.

[39] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 502–511.

[40] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proc. Conf. Program. Lang. Des. Implementation*. New York, NY, USA: ACM, 2011, pp. 329–342. [Online]. Available: https://doi.org/10.1145/1993498.1993537

[41] M. Monperrus and M. Martinez, "Conservation and replication with CVS-vintage: A dataset of CVS repositories of Java software," Tech. Rep., 2012. [Online]. Available: https://hal.science/hal-00769121/

[42] E. W. Myers, "AnO(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1–4, pp. 251–266, 1986.

[43] A. T. Nguyen et al., "API code recommendation using statistical learning from fine-grained changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. New York, NY, USA: ACM, 2016, pp. 511–522. [Online]. Available: https://doi.org/10.1145/2950290.2950333

[44] A. Panichella, "A systematic comparison of search-based approaches for LDA hyperparameter tuning," *Inf. Softw. Technol.*, vol. 130, 2021, Art. no. 106411. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300069

[45] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of Java source code," *Softw.: Pract. Experience*, vol. 46, pp. 1155–1179, 2015, update for oadoi on Nov. 2, 2018. [Online]. Available: https://hal.inria.fr/hal-01078532

[46] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[47] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[48] R. Shu, T. Xia, J. Chen, L. Williams, and T. Menzies, "How to better distinguish security bug reports (using dual hyperparameter optimization)," *Empirical Softw. Eng.*, vol. 26, no. 3, 2021, Art. no. 53. [Online]. Available: https://doi.org/10.1007/s10664-020-09906-8

[49] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2018, pp. 130–140.

[50] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019.

[51] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 483–494.

[52] L. Villalobos-Arias and C. Quesada-López, "Comparative study of random search hyper-parameter tuning for software effort estimation," in *Proc. 17th Int. Conf. Predictive Models Data Anal. Softw. Eng. (PROMISE)*. New York, NY, USA: ACM, 2021, pp. 21–29. [Online]. Available: https://doi.org/10.1145/3475960.3475986

[53] C. Yang and E. J. Whitehead, "Pruning the AST with hunks to speed up tree differencing," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2019, pp. 15–25.

[54] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3103–3116, Aug. 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3079841

[55] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2015, pp. 913–923.

Matias Martinez received the graduate degree in computer science from UNICEN, Argentina, and the Ph.D. degree from the University of Lille, France. He is a Researcher with the Universidad Politécnica de Cataluña—BarcelonaTech, Spain. He was an Associate Professor with the Université Polytechnique Hauts-de-France, France. His research focuses on empirical software engineering, automated software engineering, and software sustainability.

Martin Monperrus received the master's degree from the Compiègne University of Technology and the Ph.D. degree from the University of Rennes. He is a Professor in software technology with the KTH Royal Institute of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, AI on code, and program hardening. For more information, please visit https://www.monperrus.net/martin/.

Jean-Rémy Falleri received the "Habilitation" degree in computer science from the University of Bordeaux and the Ph.D. degree from the University of Montpellier. He is a full Professor with Bordeaux INP and a Researcher with the LaBRI Laboratory. He is also a junior member at the "Institut Universitaire de France." His research interests revolve around software engineering and software evolution.