2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms

Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, Egor Bulychev
source{d}
Madrid, Spain
{vadim,waren,hugo,konstantin,egor}@sourced.tech

***Abstract*—Source code reviews are manual, time-consuming, and expensive. Human involvement should be focused on analyzing the most relevant aspects of the program, such as logic and maintainability, rather than amending style, syntax, or formatting defects. Some tools with linting capabilities can format code automatically and report various stylistic violations for supported programming languages. They are based on rules written by domain experts, hence, their configuration is often tedious, and it is impractical for the given set of rules to cover all possible corner cases. Some machine learning-based solutions exist, but they remain uninterpretable black boxes.**

This paper introduces STYLE-ANALYZER, a new open source tool to automatically fix code formatting violations using the decision tree forest model which adapts to each codebase and is fully unsupervised. STYLE-ANALYZER is built on top of our novel assisted code review framework, LOOKOUT. It accurately mines the formatting style of each analyzed Git repository and expresses the found format patterns with compact human-readable rules. STYLE-ANALYZER can then suggest style inconsistency fixes in the form of code review comments. We evaluate the output quality and practical relevance of STYLE-ANALYZER by demonstrating that it can reproduce the original style with high precision, measured on 19 popular JavaScript projects, and by showing that it yields promising results in fixing real style mistakes. STYLE-ANALYZER includes a web application to visualize how the rules are triggered. We release STYLE-ANALYZER as a reusable and extendable open source software package on GitHub for the benefit of the community.

***Index Terms*—assisted code review, code style, decision tree forest, interpretable machine learning**

I. INTRODUCTION

The way source code is formatted has a significant impact on both comprehensibility and maintainability [1], [2], and ensuring that a codebase is consistent in style is tedious and time consuming. When human reviewers have to consider formatting during a code review session, their ability to spot defects and bugs is diluted by too many concerns [3]. At Google, software engineers who obtain *readability* certification for a specific language are assigned for approval during the review process to ensure stylistic consistency across the codebases [4]. However, those developers in charge of a particular codebase might rotate on a regular basis, and some projects might not even have a linting process configured early on. Furthermore, different programmers often have different code formatting preferences, such as indentation, white space usage, or brace positioning. This eventually leads to variations of the resulting code style. Of course, there are plenty of configurable linting tools for source code, whether included into IDEs like INTELLIJ or external applications like ESLINT for JavaScript. However, configuring the style checks is not obvious; these tools can be too opinionated and hard to set up to satisfy team's wishes. Furthermore, there may be dependencies between options, and the tools struggle to take the context information into account [5]. Finally, linters do not always suggest fixes for the violated rules, and when they do, applying those fixes might not be convenient for programmers.

In this paper, we introduce STYLE-ANALYZER, a tool to solve the automatic code formatting problem at code review time, suggesting changes to pull requests on GitHub when style inconsistencies are detected. An example of such a suggestion is given in Figure 1. We include this solution into our new assisted code review framework, LOOKOUT, that allows running pluggable code analyzers over pull requests. Thus our training set is restricted to a single repository. In order to make the suggested changes interpretable and establish trust with the users, we employ an adaptive machine learning model which learns code formatting rules from existing code. We gather the implicit and explicit user feedback on the triggered rules and have the ability to disable the misbehaving ones while leaving the rest intact. Finally, the analysis performance is high enough to yield results with small time footprint - under five minutes for large pull requests. To satisfy our requirements and limitations, we first define the underlying code style of a repository in terms of language models, then train a decision tree forest on the AST-augmented token stream. Finally, we extract rules from the trees and optimize them. Debugging such a complex pipeline requires dedicated tooling, so we developed a benchmarking suite and an application to visualize how the rules are triggered.

The main contributions of this paper are:

- STYLE-ANALYZER, an analysis running on LOOKOUT that mines interpretable code formatting rules using machine learning, validates new code against them, and suggests fixes when appropriate.
- A new assisted code review framework, LOOKOUT, which watches GitHub repositories and triggers a set of analyses when pull requests are updated or new commits are pushed. LOOKOUT reports the result of these analyses as GitHub comments to pull requests, leveraging the recently appeared *GitHub Suggested Changes* feature. LOOKOUT allows for rapid development of new analyses

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00073

and provides a universal code parsing API.
- A web application which annotates the analyzed source code with the extracted features of code tokens and the triggered rules.

The rest of the paper is organized as follows. Section II revises prior work related to fixing formatting defects. In section III, we explain the model behind STYLE-ANALYZER. Section IV presents LOOKOUT, our framework for assisted code review and details how it helped to implement STYLE-ANALYZER. In section V, we detail the evaluation of our model. Section VI first discusses the current limitations of STYLE-ANALYZER approach and then explains how they can be addressed in future works.

II. RELATED WORK

Some machine learning approaches have already been explored to suggest code improvements for stylistic consistency. Allamanis *et al.* were the first to introduce the coding convention inference problem. They built NATURALIZE, a language-agnostic code formatting suggester that learns formatting conventions and generates rules from them [6]. NATURALIZE proceeds in two steps: first it generates formatting candidates, second it ranks those candidates for later use in development tooling if they meet a quality threshold. Nevertheless, NATURALIZE considers only the local context and can produce semantically disruptive suggestions; thus it is not fully automatic. More recently, CODEBUFF — an automatic code formatter for Java, SQL and ANTLR — was proposed [7]. It exerts machine learning to obtain abstract formatting rules from a representative corpus. However, CODEBUFF needs to be configured by example and covers neither mixed indentation (tabs vs. spaces) nor mixed quotes (single vs. double). Besides, CODEBUFF uses a *k*-Nearest Neighbor machine learning model to classify the unknown feature vectors, which makes its decisions harder to understand by humans.

Another closely related group of tools encompasses rule-based *default formatters*. In contrast to STYLE-ANALYZER that learns the existing style of a codebase, they are often opinionated and enforce a specific subset of styles approved by their designers: if a user is targeting a specific formatting style, those tools might prove impossible to setup. Among the most popular ones, we can point out GNU INDENT[1] and PRETTIER[2] for example.

Other approaches address slightly different albeit related issues. As an illustration, Wang *et al.* have developed a heuristic solution to automatically insert blank lines in methods to improve their readability [8].

III. METHODOLOGY

STYLE-ANALYZER learns the underlying code formatting rules of a repository in a completely unsupervised manner — and with zero prior domain knowledge — from the supplied files constituting a specific Git revision. STYLE-ANALYZER

[1]https://www.gnu.org/software/indent/
[2]https://github.com/prettier/prettier

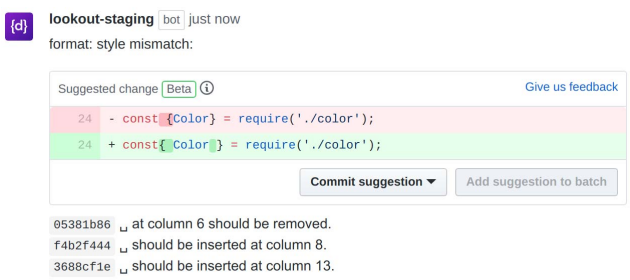

Fig. 1: Example of LOOKOUT comment output from STYLE-ANALYZER and published as a *Github Suggested Change*.

is built on the concept of a holistic language model of source code. Language models have been largely explored to calculate various probability distributions of source code using maximum likelihood estimates [9]. We do not rely on any explicitly labeled data; instead, we model the formatting code tokens from the surrounding context where they occur in the files. We further model the absence of the formatting tokens in the rest of the relevant places (in-between semantic tokens). The learned language model predicts whether new, proposed code changes follow the established conventions.

A. Code style and feature extraction

We represent a source code file as a linear sequence of tokens. An example of representation is given in Figure 2. Some of these tokens correspond to the nodes in the AST representing the file, while others are inserted to mirror whitespace characters, keywords, quotes, and braces — in other words, auxiliary tokens which are absent from the AST but allow the accurate reconstruction of the original file. The exact set of the latter depends on the programming language and the AST structure; we stick to JavaScript and use BABELFISH [10] to obtain ASTs with a structure that is common to many languages. We will refer to those ASTs as UASTs for *Universal Abstract Syntax Trees*. The ability to reproduce the exact source code from the parsed stream of tokens is a prerequisite.

Regarding the formatting elements which we predict, we consider 9 atomic classes listed in Table I. To fit our sequence prediction framework, those classes are combined to form compound sequences, e.g., four spaces indentation increase ($␣^+␣^+␣^+␣^+$) or two consecutive newlines (↲ ↲).

We take advantage of the sequence prediction framework to correctly model insertions and deletions. From a given sample, the prediction to a larger sequence can model insertion, to a smaller sequence, deletion. To allow insertions from places where there is no formatting and full deletions from places where formatting is present, we include the empty sequence as a label.

The filtered labels are one-hot encoded, and we solve a classification problem. To extract features from the formatting elements, we mix sequential and structural information by combining 2 different representations of source code [11]. We explore a symmetric window of fixed size in the stream of tokens, as well as climb a few levels up the AST hierarchy.

1.	␣	whitespace
2.	↦	tabulation
3.	↲	newline
4.	$␣^{+}$	whitespace indentation increase
5.	$␣^{-}$	whitespace indentation decrease
6.	$↦^{+}$	tabulation indentation increase
7.	$↦^{-}$	tabulation indentation decrease
8.	'	single quote
9.	"	double quote
10.	∅	empty gaps between non-label nodes, NOOP

TABLE I: List of atomic label classes

As described in Figures 2 and 3, the window that we adopted in our experiments was 5 nodes to the left, 5 nodes to the right and 2 AST parents up. A token's parent here means the *Lowest Common Ancestor* (LCA) of the closest token's sibling in the UAST — in other words, the deepest UAST node that has both the closest left and right UAST nodes as descendants. To compute the LCA, we trace the UAST two times respectively from the two siblings down the root, and return the common node just before the encountered mismatch.

Universal Abstract Syntax Trees provide the following node attributes:

Value the content of the corresponding token, if it exists, otherwise an empty string.

Internal type the string which indicates the type of the node in the native AST — that is, before the UAST conversion.

Roles the set of strings which indicate UAST node roles. There is a list of supported roles in the official BABELFISH documentation with around 200 elements, e.g. `Identifier`, `Literal`, `Comment`.

Start position the line number and the column number where the node begins. It matches the token start position if the node is an AST leaf, otherwise, it is defined recursively as the minimum start position among its children.

End position the line number and the column number where the node ends. It is defined similarly to the start position.

Hence, we decided to record the following features for each formatting element:

Label matches the one-hot encoded label if the token is a formatting sequence. It is collected only for the nodes to the left of a label.

Internal type one-hot encoded. If there is no corresponding UAST node, all zeros.

Reserved index one-hot encoded — for those tokens with an empty internal type, we collect all possible token values in the analyzed files.

Length the length of the token value.

Roles a fixed length vector with "1"-s at the indexes of the relevant roles. Again, if the token is not backed up with a UAST node, it is all zeros.

Position information collected only for the nodes to the left of a label. It is all zeros for the nodes to the right.

- File offset difference with the previous node in the sequence.
- Column number difference alike.
- Line number difference alike.

Regarding the parents, we extract their internal types and roles. Besides, we record the start position of the predicted node, namely the line and column numbers.

One of the challenges of designing an unsupervised predictive model is to not leak the information about the labels to the engineered features. It is easy to miss some obvious relations between the context and the label which are perilous for an unbiased reasoning at test time [12]. For example, we could include the difference in offsets between the immediate left and right neighbors as one of the features. This difference indicates the exact length of the predicted token in-between and the model inevitably overfits to it. As a result, it becomes limited to suggestions of the same length, e.g. it can no longer predict zero-length NOOPs instead of non-empty formatting tokens. We mitigated the aforementioned perils by selecting different features for the left and the right token siblings. More precisely, the right token features are limited to semantic-only information to avoid any formatting information leakage.

The overall size of our feature vector sums to over 4000. We apply univariate feature selection [13] to keep the 500 best features according to the ANOVA F-value criterion.

The number of features to compute grows rapidly with the size of the analyzed repository to reach tens of millions of floating point values for a large repository, and the feature extraction step becomes the main pipeline bottleneck. It is therefore important to tune its performance. We took advantage of the efficient sparse data structure operations in the SCIPY [14] library to avoid unnecessary computations. We saturate the size of the training set to guarantee the strict run time bounds. It is limited to the first 2 MB of concatenated randomly shuffled files. We additionally filter out files that are automatically generated, e.g. minified, by the maximum allowed line length of 500.

B. Decision Trees

Given the extracted features, we opt for decision trees as our machine learning model. This choice is motivated by our interpretability — "white box" — requirement. Decision trees naturally and transparently explain their predictions by following the respective branches [15]. Decision trees group examples seen in the training set into different leaves which are created to minimize the diversity of examples in each leaf. A single decision tree is unlikely to achieve good predictive power on plenty of data; hence, the common practice is to train several decision trees and combine their predictions. There are two widespread approaches: ensembling (decision tree forest) and boosting (gradient boosted decision trees, GBDT). The random tree forest algorithm [16] better suits our requirements since GBDT are hard to interpret due to tree chaining. We use the `RandomForestClassifier` implementation from SCIKIT-LEARN [17].

```
∅function␣classesToArray∅(␣value␣)␣{↲
↦+  if␣(␣isArray∅(␣value␣)␣)␣{∅return␣value∅;∅}↲
    if␣(␣typeof␣value␣===␣"string"␣)␣{↲
↦+      return␣value∅.∅ match(␣rnothtml␣)␣||␣[]∅;↲
↦-  }↲
    return␣[]∅;↲
↦-}∅
```

Fig. 2: Example of JavaScript code with style inconsistencies. The source code is annotated according to our code representation. The green labels fit the model's predictions while the red ones differ, and the highlighted labels correspond to the 10-token window used for feature extraction around the first red sample.

y : ∅

ŷ : ↲↦+

Confidence: 0.975

Support: 3230

Attributes

`-1.value = {`
`-3.value` not in {`else`}
`+1.internal_type` not in {`StringLiteral`}
`+2.roles` not in {`LITERAL`, `COMMENT`}
`^1.roles` in {`IF`, `STATEMENT`}

(a)

Features

root: {} 4 items
left: [] 5 items
node: [] 1 item
parents: [] 2 items
1: {} 2 items
internal_type: IfStatement
roles: {IF, STATEMENT}
2: {} 2 items
right: [] 5 items

(b)

Fig. 3: (a) Prediction of our model and insight into the triggered rule. The window is centered at the highlighted red label in Figure 2. The rule's details include the its confidence and support. (b) Features extracted at the same location in the code. For example, the first parent in the UAST hierarchy has the following two roles: `If` and `Statement`.

We run hyper-parameters optimization during the training phase to maximize the accuracy metric. More precisely, we perform 100 iterations of Bayesian optimization using Gaussian Processes [18] with stratified 3-fold validation to optimize the following hyper-parameters:

Model Whether to train a random forest or a decision tree;
Depth Authorized maximum depth for the trees;
Considered features Features considered for each split during tree construction;
Minimum samples per split Minimum number of samples to split a node during tree construction;
Minimum samples per leaf Minimum number of samples in a leaf during tree construction;

100 iterations were always enough to converge for all the repositories in our evaluation dataset from subsection V-A.

After obtaining a trained decision tree forest, we transform it to production rules [19]. According to the established terminology, we call feature comparisons along a branch attribute comparisons. Each path from the root of a tree to one of its leaves is a set of attribute comparisons which leads to a specific label prediction. We say that such a set of attribute comparisons is a rule. Each rule has a training precision value, *i.e.* the number of times it predicted correctly divided by the number of times its attribute comparisons were simultaneously true. We call this value the confidence of a rule.

Having extracted the rules from the decision tree forest, we simplify them and reduce their number so that they are easier to comprehend. First we set a confidence threshold and remove all the rules which are not precise enough. If this threshold is close to 1, we end up with a small number of rules which are very precise but cover only a tiny fraction of the examples. In contrast, if this threshold is close to 0, we keep all the rules, staying imprecise but retaining the best recall.

In order to simplify the rules, we merge all comparisons of the same attribute together, therefore doing at most two comparisons per attribute.

Afterwards, we perform attribute pruning — removing parts of the rules that are redundant and suppress generality. For each rule, we collect the sets of samples on which attribute comparisons *incorrectly* predict the corresponding label. We further build the undirected similarity graph of those sets. It's vertices are attributes, and each pair is connected if the Jaccard similarity between their sample sets is bigger than a certain threshold (we chose 0.98). Finally, we perform Multilevel community detection **??** on that graph and greedily leave only a single representative of each community. The described pruning algorithm is computationally expensive, so we do not run it while optimizing the model hyper-parameters.

The last stage is removing duplicated rules after attribute pruning. There also exist techniques to globally optimize a

collection of rules [19] but we found them impractically slow for our task.

The resulting set of production rules comprises the final model and is both considerably simpler than the raw decision trees, more accurate and works better with samples outside of the training set without sacrificing the recall.

C. Applying the rules

Application of the rules to changed code in pull requests has several points worth noting. We create suggestions for every prediction that does not match the factual label. It is possible for more than a single rule to fire — in such a case we use the most confident one. It may also happen that no rule fires — then we silently proceed to the next sample.

There are paired format tokens, and they are processed independently according to our scheme. Hence, there can be two predictions which contradict each other. For instance, we may suggest a string literal with different left and right quotes. We have to handle this special case separately by respecting the most confident rule and changing the paired prediction accordingly.

Besides, we check that the proposed fixes do not break the code. For example, the model can remove a whitespace character in the middle of an expression and thus change the AST, as shown in Figure 4. We parse a new UAST for each changed line and compare it with the original UAST. If any difference is detected, we drop the suggestions on the matching line. However, this check is costly if there are many suggestions, and we leverage a parsing optimization: we avoid parsing a complete file each time. Instead, we take the common parent UAST node of the checked line and parse the code block it spans over. The parsing may fail — some nodes are not parsable in isolation; we then continue to climb up the AST hierarchy until we succeed. On average, this optimization brings a 1.5x speedup. The AST checks are independent of each other, so they can be efficiently parallelized to leverage multiple CPU cores. However, they execute in Python which has a Global Interpreter Lock, so the parallelization yields only a moderate 50% run time decrease on a 32-core machine — due to the large increase in computing power needed to obtain this reduction, we turn it off by default. Similarly to training, we limit the size of changed files that we analyze in a pull request. Only the first 2 MB are analyzed.

STYLE-ANALYZER finishes analyzing a pull request by generating code for the suggestions. The main challenge here is to properly handle the indentation changes. Given a sequence of altered lines, we may miss indentation fixes for some of them — that is, we might create indentation conflicts. There are two strategies:

1) Independently generate code according to the predictions, ignoring the potential conflicts;
2) Maintain a consistent state by also modifying the other impacted indentations for each prediction — *e.g.*, increase the indentation of a whole block for a single prediction.

The second strategy can be more accurate but would require a model that makes dependent predictions (such as a recurrent neural network or a hidden Markov model). STYLE-ANALYZER makes independent predictions for each sample, as in the first approach.

```
   function classesToArray( value ) {
      if ( isArray( value ) ) {
         return value;
      }
      return [];
   }
```

(a)

```
∅function␣classesToArray∅(␣value␣)␣{↲
↦+   if␣(␣isArray␣(␣value␣)␣)␣{↲
↦+       return∅value∅;↲
↦-   }
     return␣[]∅;↲
↦-}∅
```

(b)

Fig. 4: (a) Original example of valid JavaScript code (b) examples of predictions of formatting elements that are changing the AST (in blue) and thus rejected during the AST filtering.

D. User feedback

We hash each rule's body — the attribute comparisons and the predicted label — to a 32-bit unsigned integer. Those hashes are prepended to fix descriptions in GitHub comments, so users can identify which rules trigger. Users have an ability to disable the rules which are wrong or too noisy in their opinion by blacklisting the identifiers in the LOOKOUT configuration file that resides at the repository root. Blacklisted suggestions are never shown.

Furthermore, LOOKOUT provides information about which suggestions have been merged and which have not. We measure the ratio of unmerged suggestions over all suggestions for each rule. Once the number of unmerged suggestions exceeds 10 and the ratio drops below 0.9, the rule gets blacklisted.

If we train another model, new rules may emerge. According to our calculations, a different random seed yields 75% of unique rules and that makes our blacklisting mechanism less efficient. We mitigate this negative impact by only training new models each 100th commit. Better rule matching approaches can be explored in the future.

IV. IMPLEMENTATION

A. LOOKOUT: an assisted code review framework

STYLE-ANALYZER runs its analysis when pull requests are updated on GitHub. There are other options to assist software developers: in an IDE while they type, in a Continuous Integration (CI) script or running analysis periodically as asynchronous scheduled jobs. We discarded them for different reasons. First, IDE solutions are expected to produce results

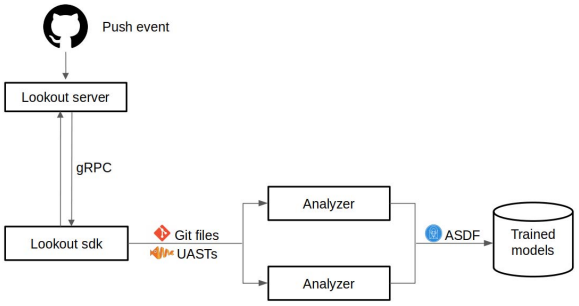

Fig. 5: The flow of a *Push* event

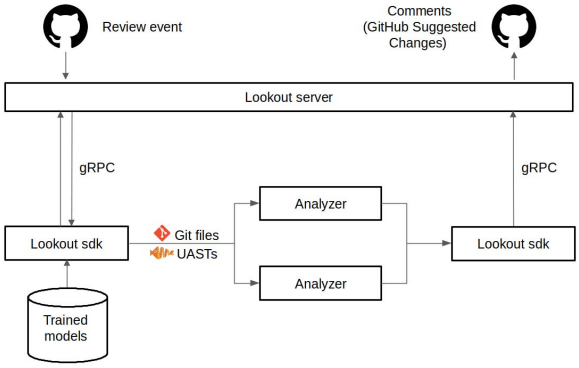

Fig. 6: The flow of a *Review* event

instantly on developers' hardware and therefore enforce strict run time, memory and CPU utilization constraints. Pull request analysis may run in the cloud on well-defined and powerful hosts and it is not required to generate results in few seconds, thus is potentially capable of more sophisticated assistance. Second, CI does not have a good user interface for code suggestions. In fact, the only way to report anything in CI is to print to standard output stream, whereas GitHub enables single-click fix merges as can be seen in Figure 1. Third, our tool should remain a part of the development workflow, otherwise there is a tendency to ignore or bypass the numerous accumulated suggestions. This has been demonstrated at Google scale by Sadowski *et al.* [20]. Hence we abandon the scheduled jobs option. Running the analysis during code review also has its drawbacks: the code review feedback loop is longer compared to IDEs because the main way to discover fixes is to create a pull request and look at STYLE-ANALYZER's comments.

The purpose of the LOOKOUT framework is to deliver assisted code review to everybody in an easy-to-setup, easy-to-use, easy-to-extend fashion. It contains the server application which listens to repository events from GitHub. We call it the LOOKOUT server. It is possible to create applications which register with the LOOKOUT server to run code analyses. We call them LOOKOUT analyzers. Whenever new commits are pushed or pull requests are updated, the LOOKOUT server communicates with the registered analyzers. In case of a push event (see Figure 5), that communication is just a notification for analyzers to update their internal state, if they have one. In case of a pull request event (see Figure 6), the LOOKOUT server obtains a list of review suggestions from each of the analyzers. It aggregates the lists and posts comments to the corresponding lines in the files of the pull request.

If an analyzer suggests better code, it can format the Markdown text of the corresponding GitHub comment in a special way to leverage *GitHub Suggested Changes*. Released in October 2018, that feature provides a user interface to accept or reject proposed code edits with a single mouse click. The LOOKOUT server inspects which code suggestions have been merged and supplies this information to analyzers, hence establishing a valuable user feedback loop.

B. LOOKOUT *architecture*

All the communication happens through gRPC remote procedure call interfaces. The LOOKOUT server and each of the analyzers run as gRPC services. This allows to scale easily and ensures solid isolation between the components. The LOOKOUT server is written in the Go programming language and is available online[3]. Thanks to gRPC, there are no limitations on languages or frameworks for analyzer implementations.

The LOOKOUT server does not provide the contents of the changed files to analyzers by default. Instead, it serves another gRPC endpoint for data retrieval. It is possible for an analyzer to request the contents of any file at any revision in one of the watched Git repositories from that endpoint. Besides, the LOOKOUT server is integrated with BABELFISH, the platform for language-agnostic source code parsing [10]. BABELFISH parses code written in supported programming languages into Universal Abstract Syntax Trees (UASTs). A UAST has the standard annotated format of the parsed syntax tree and simplifies cross-language code analysis by providing language-agnostic annotations (e.g. function, identifier) on the corresponding sub trees.

LOOKOUT abstracts analyzers from dealing with the GitHub API, working with Git, parsing code, and discovering user feedback, thus allowing to focus on code analysis problems.

C. Software Development Kit

LOOKOUT offers a Software Development Kit to ensure rapid development of new analyzers. There are two flavors of the SDK: low-level and high-level. The low-level SDK makes no assumptions on the nature of analyzers and serves as a thin proxy layer over gRPC. The high-level SDK is based on the low-level one. It is written in Python, is also available online[4] and targets stateful analyzers. Statefulness means here that a push event leads to a change of the analyzer's state which needs to be persisted. Git repositories typically contain more than one branch. The LOOKOUT server notifies about pushes to each branch, and it is the responsibility of analyzers to

[3]https://github.com/src-d/lookout
[4]https://github.com/src-d/lookout-sdk-ml

correctly handle them. The typical behavior would be ignoring all the branches but the one which is marked as main on GitHub. In most of the cases, it is the `master` branch, so we exercise `main` and `master` interchangeably below.

The high-level SDK takes care of technical issues, such as:

Work with gRPC: load balancing, connection pooling and related threading constraints;

Maintenance of the database with analyzer states: the obvious way to achieve persistence. The states are stored in Advanced Scientific Data Format [21] with LZ4 compression on disk. The metadata about the states lives in any SQL database that is supported;

Caching states: some repositories are going to send events much more frequently than others, and each database operation has a cost;

Logging: logs include the request context with repository URL, revision, type of event, etc;

Metrics collection: requests per time unit per analyzer type, elapsed time, various repository statistics, and failures. Metrics collection is especially important for machine learning-driven analyzers, because it is critical to measure various quality metrics at runtime to understand and improve the underlying models;

Common file filters: removing certain files from consideration. Examples of such filters are binaries; very long lines (e.g. minified JavaScript); blacklisted file name prefixes (`node_modules` in JavaScript, `vendor` in Go);

Common operations with file contents and UASTs: *e.g.* determining changed line numbers for a pair of files.

The high-level SDK defines the API for designing analyzers in Python. It is enough to implement two functions — `train` and `analyze` which are invoked respectively on push and pull request events — to harness a fully featured analyzer. For example, a basic analyzer that corrects typos in identifier names requires fewer than 100 lines of code[5].

The high level SDK was chosen to create STYLE-ANALYZER — the analyzer which trains on the master branch of a repository and suggests code formatting fixes. STYLE-ANALYZER's source code is available on GitHub[6] together with the related documentation and the datasets built for evaluation.

V. Evaluation

We evaluate STYLE-ANALYZER on two benchmarks. The first aims to measure how well STYLE-ANALYZER models the style of a repository by applying a trained model on a held-out test dataset. The predictions of the model are considered correct if they match the actual formatting elements in the original source code. We carry out this evaluation on a collection composed of 19 top-starred repositories from GitHub. We refer to that benchmark as the style modeling benchmark and report on the related findings in subsection V-A. The second benchmark approximates the performance of STYLE-ANALYZER in the wild by measuring how well it fixes style mistakes that we seeded manually in two large repositories. We refer to it as the style defects fixing benchmark and devote to it subsection V-B.

We trained all the models on a 32-core machine with 128 GB of memory running on Linux. In practice, each model requires less than 5 GB of memory to be trained and less than 1 GB to be applied. The pull request analysis always finishes in less than 15 min.

A. Style modeling benchmark

We select 19 top starred open source GitHub repositories which have JavaScript as their main language and ensure the diversity of their sizes to study how STYLE-ANALYZER performs on both big and small codebases. The head revisions of the selected repositories correspond to January 2019. Among the largest repositories are `nodejs/node` and `facebook/react-native`, which have more than one million lines; see Table II for further statistics.

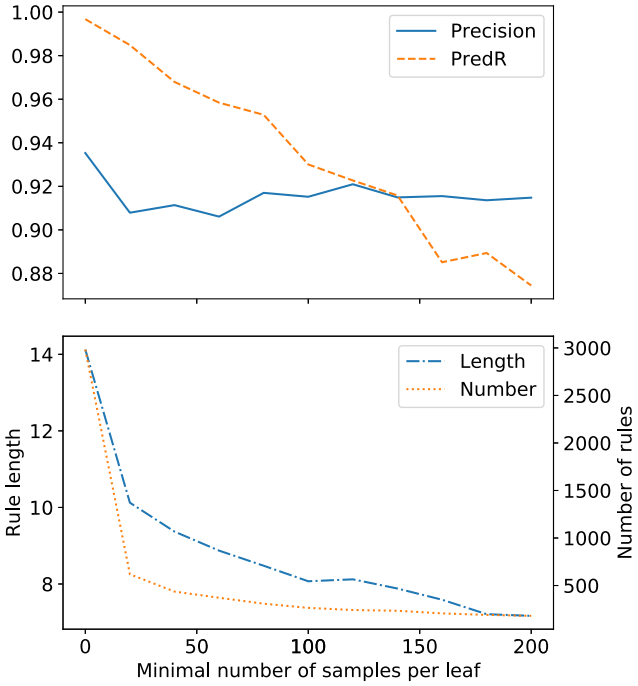

Fig. 7: Impact of minimum samples per leaf on performance.

We make an assumption that the code style is consistent on file level, and we randomly split each repository's files into two groups. The size in bytes of the first group is 80% of the overall size, of the second is 20%. The files from the first group are used to train the model while the files from the second group are used for the validation benchmark.

To evaluate STYLE-ANALYZER, we focus on both precision and prediction rate (PredR). Precision indicates how noisy the predictions are. We believe that a score significantly lower than 95% would jeopardize the users' trust in the model. PredR, in turn, points out how often the model makes predictions. Since we evaluate STYLE-ANALYZER on real projects, it should

[5]https://github.com/src-d/lookout-sdk-ml/blob/master/lookout/core/examples/typos.py

[6]https://github.com/src-d/style-analyzer

repository	precision	PredR	recall	f1	train samples	LoC	unique labels	rules	avg. rule length	training time, min
node	0.965	0.951	0.918	0.941	374,298	1,074,016	26	641	13.8	160
webpack	0.957	0.956	0.915	0.936	358,012	77,731	18	666	11.6	154
meteor	0.900	0.845	0.761	0.825	337,627	235,411	33	557	12.1	287
react	0.943	0.974	0.919	0.931	304,465	170,920	18	780	11.7	115
atom	0.955	0.995	0.950	0.952	265,521	137,599	16	440	11.1	125
react-native	0.940	0.962	0.904	0.922	264,961	131,192	22	693	13.7	206
jquery	0.972	0.959	0.933	0.952	197,072	55,384	19	391	9.5	82
storybook	0.940	0.953	0.896	0.917	161,366	43,757	15	494	10.3	39
freeCodeCamp	0.928	0.960	0.891	0.909	114,020	29,044	14	474	9.8	35
express	0.937	0.979	0.918	0.928	78,411	17,460	10	269	9.6	15
30-seconds-of-code	0.951	0.977	0.930	0.940	67,737	11,813	10	151	7.1	8
evergreen	0.894	0.958	0.857	0.875	38,387	24,507	19	66	11.2	25
citgm	0.936	0.933	0.873	0.904	21,941	5349	12	14	6.0	4
axios	0.940	0.951	0.895	0.917	21,130	7342	10	143	7.3	4
create-react-app	0.895	0.862	0.772	0.829	16,718	14,489	12	101	6.8	4
redux	0.937	0.844	0.791	0.858	14,783	8963	13	25	6.5	5
reveal.js	0.897	0.842	0.755	0.820	9974	12,926	14	32	8.6	2
carlo	0.878	0.931	0.817	0.846	5529	3449	8	78	6.2	2
telescope	0.806	0.570	0.460	0.585	731	467	5	2	2.0	1
average	0.925	0.916	0.850	0.884	139,615	—	15	317	9.2	—
weighted average	0.943	0.947	0.894	0.918	—	—	—	—	—	—

TABLE II: Metrics measured on the validation part of the dataset. The last row is weighted by the number of samples.

be mentioned that inconsistent formatting across repositories could lead to poor precision. Besides, the model does not always output a prediction for each labeled token because we remember that some rules can be disabled, see subsections III-B and III-C. That is why we measure the percentage of predictions made:

$$\text{PredR} = \frac{\text{number of predictions}}{\text{number of samples}}$$

In practice, the label frequency distribution has a long tail. Thus we measure the frequencies of compound labels and exclude the rare ones by putting a threshold of 80 occurrences throughout all the files. According to our measurements, this value retains approximately 30% of the labels and reaches a decent compromise between keeping significant labels and cutting away outliers. Regarding the confidence threshold whose goal is to select the most relevant rules, see subsection III-B, we experimentally found 0.92 to yield good results; it keeps from 10% to 50% of the rules depending on the dataset and model. To further reduce the number of rules — and to regularize the model — we set a minimum number of samples for each leaf of the tree to a threshold that we determine experimentally. Figure 7 shows the evolution of the precision and PredR on one side, and the number of rules and their average length on the other side, both over the minimum number of samples per leaf, on the training set. The number of trees in the random forest was fixed to 10. After analyzing the results, a good trade-off between high PredR and the interpretability of the rules is to set the optimal value of the minimum number of samples per leaf to 80; we will keep this value in the next experiments.

Table II gathers the model's metrics for the 19 top starred JavaScript repositories on which we achieve 94% precision at 95% PredR on weighted average. STYLE-ANALYZER performs poorly on small codebases, as can be observed in Figure 8. We relate this fact to not having a sufficient number of training samples. Furthermore, the precision on `evergreen` is low because it contains a mixture of JavaScript and JSX with different formatting.

We also note that repositories with strong style guidelines — such as `jQuery` — are easier to model and produce simpler models (with less rules) by a significant margin compared to repositories with inconsistent styles — such as `FreeCodeCamp`.

We have additionally measured the ratio of rules which contain parental attribute comparisons, i.e. leverage the structural information: 60%-90% depending on the repository, which proves its usefulness. UAST difference check cuts 8% predictions on average across 8,000 files, which grants a considerable precision gain. Attributes pruning removes up to 60% comparisons in our 19 repositories, which leads up to 55% less rules without any precision drop.

The style modeling benchmark gives a general idea of how good the model can approximate the style of a repository. It does not directly demonstrate the quality of suggestions for

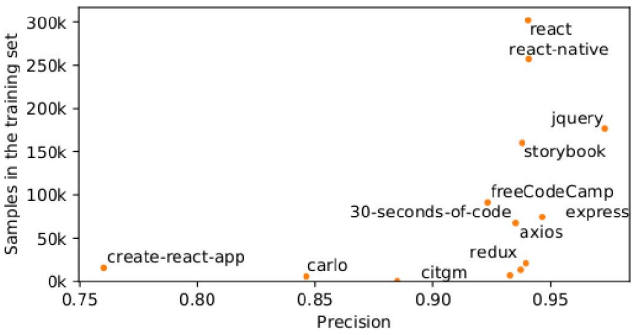

Fig. 8: Effect of the number of samples in the training set on precision.

```
function classesToArray( value ) {
  if ( isArray( value ) ) {
     return value;
  }
  if ( typeof value === "string" ) {
    return value.match( rnothtml ) || [];
  }
  return [];
}
```

(a)

```
function classesToArray(value) {
  if ( isArray(value) ) {return value;}
  if ( typeof value === 'string' ) {
     return value.match(rnothtml) || [];
  }
  return [];
}
```

(b)

Fig. 9: (a) Example of JavaScript code and (b) its modified version that includes style inconsistencies: (i) spaces around API calls have been removed (ii) a tabulation has been replaced by a four spaces (iii) the first `if` statement has been shortened into one line (iv) single quotes have replaced double quotes

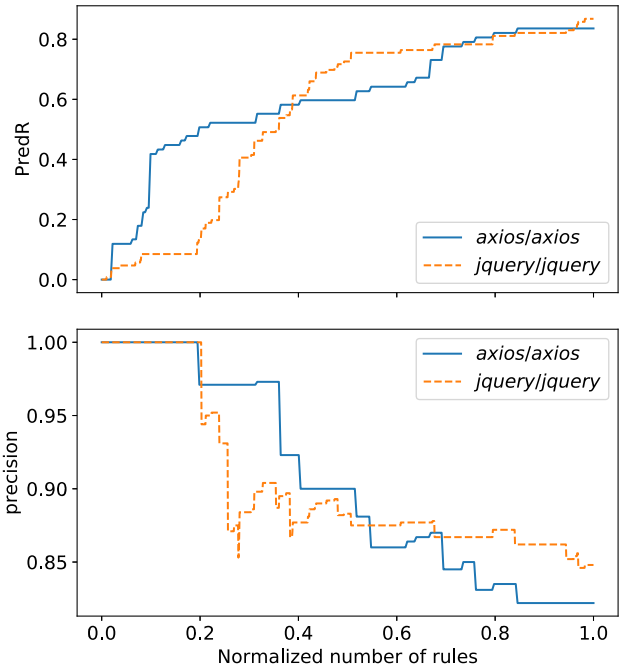

Fig. 10: Metrics variance on the dataset of style defects: (a) PredR and (b) precision evolutions over the number of rules retained by the model. The rules are sorted by confidence and their overall amounts are respectively 183 and 480 elements for the *axios/axios* and *jquery/jquery* GitHub repositories.

fixing style defects in source code.

B. Style defects fixing benchmark

To simulate STYLE-ANALYZER's real usage, we manually compiled a dataset of artificial style mistakes in the form of commits that deliberately introduce formatting inconsistencies in JavaScript files of some GitHub repositories. To exploit this dataset, we first train the model on the original revision and then apply those commits on top and study how the mined rules perform. We add a single format distortion per file to simplify the fix correctness check (see Figure 9). There are 80 whitespace and 60 newline insertions and removals, 10 indentation changes and 20 different pairs of quotes; 170 changes overall.

We sort the rules by confidence in decreasing order and watch quality metrics as we successively enable less and less confident rules. Incrementing the number of rules understandably increases the overall number of predictions. Yet the number of mistakes grows, too. Therefore, we are interested in the precision and PredR metrics: (i) precision equals the ratio of correctly predicted fixes over the total number of predictions made, and (ii) PredR equals the ratio of the total number of predictions made over the ground truth number of added mistakes. The results are shown on Figure 10.

We use the style defects fixing benchmark to determine the lowest rule confidence threshold to stay above the target 95% precision. The plots show that 95% precision corresponds to roughly 40% of all the rules and 60% PredR. That is equivalent to 92% rule confidence threshold. The latter number is less than 95% because more confident rules tend to trigger more frequently.

C. Rules visualization

We developed a web based application to visualize how a trained model works. This application enables users to load any JavaScript file, apply the mined rules and inspect each triggered position in the code (see Figure 11). We support changing some of the rules' optimization parameters, such as the confidence and support thresholds. The triggered rule is displayed on the right of the visualizer's window as a logical conjunction of attribute clauses. We also print its confidence and support, *i.e.* how many times the rule has been fired during training. Color coding is applied to make it easier to read a snippet of code with the corresponding model's predictions. More precisely, we use the following colors:

- the model's prediction and the input label match
- the model's prediction and the input label differ
- the model is disabled on this sample

This web application turns out to be particularly useful when debugging the models — when we need to understand their decisions or to unveil problems during feature extraction. However, it is not able to show the actual generated source code for now.

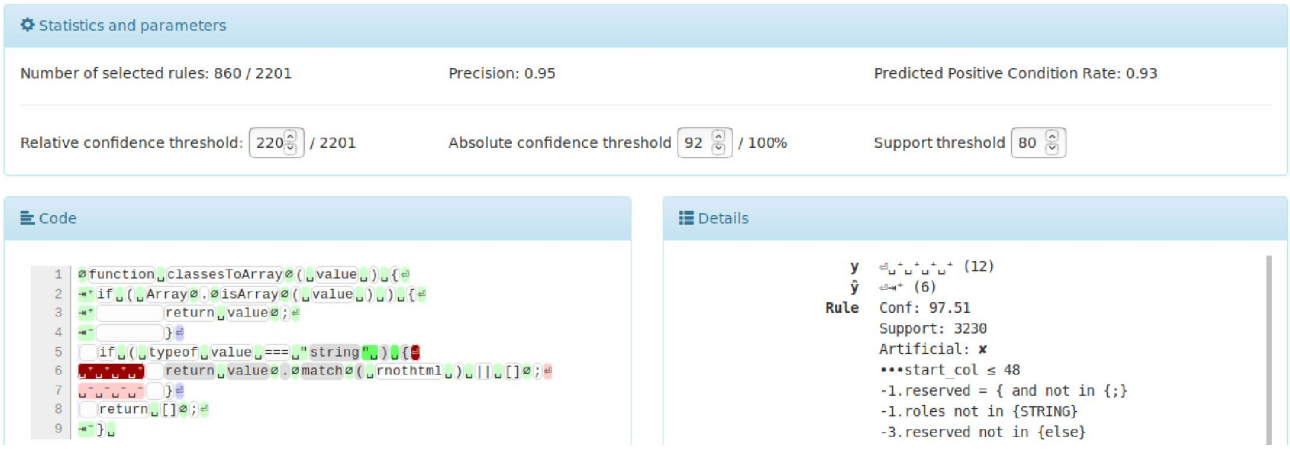

Fig. 11: A screenshot of the web based application to visualize any annotated source code and the triggered rules.

VI. Future work

As shown by the evaluation performed in section V, the formatting style can differ across a single repository. The way the current model works is to average all the examples and infer a unified solution. A promising alternative is to cluster files which are written in a similar style together and train several models on those clusters. Choosing the right model for new files would then be a challenge. A possible solution would be to apply all of them and take the least contradictory.

It would also be beneficial to present the mined rules with source code. Then the model becomes directly editable. Similar attribute comparisons should be grouped together, however, finding the optimal grouping imposes a separate problem. Besides, we would have to recompute the confidence of the edited rules.

STYLE-ANALYZER does not require any configuration, so it can be run on many repositories at scale. It would be interesting to study the common mined code formatting patterns across GitHub, although that would require improving the rule matching described in subsection III-D. Moreover, it would be possible to transfer style, for example, train formatting rules on a set of trusted codebases which follow best practices and apply them to other projects. Apart from the global style transfer, partial formatting should be explored as well.

It is promising to investigate Meta-learning approaches to training in the future [22]. Meta-learning can be summarized as principled methods that exploit meta-knowledge to obtain efficient models and solutions by adapting machine learning and data mining processes. The training would be divided into two distinct stages. The first stage and also the most time-consuming one would be to train style embeddings by asking a model to reconstruct the formatting of a codebase with only ASTs and a style embedding at its disposal. The second stage would be to adapt this knowledge to model the style of a particular repository: this would entail finding the optimum embedding to construct the formatting of the existing codebase given its ASTs. Since the embeddings are the entities in which the model condenses all the style information, it becomes possible to exploit well-known techniques from both Natural Language Processing and Computer Vision areas to find the best embeddings to model the desired style [23]. The speculated two-stage training process is sound because the model architecture can be arbitrarily complex and expressive. It can potentially learn how certain styles are applied in certain corner cases in millions of repositories, and produce predictions for a single repository, since the core of what varies is shared.

VII. Conclusion

We presented STYLE-ANALYZER, a novel open source tool that automatically fixes code formatting defects during code reviews. It learns human-readable rules of formatting from a codebase using the decision tree forest model and does not require any human participation during its training. Application of the mined rules to code in pull requests generates code suggestions which improve stylistic consistency. The achieved model performance reached the level of practical usability. Our experiments with popular JavaScript repositories on GitHub showed that STYLE-ANALYZER stays effective at diverse training set sizes, reaching both a significant percentage of predictions made (PredR) and a high precision score. Besides, we pointed out the positive performance of our tool on a manually created dataset of formatting style mistakes. We additionally demonstrated the web application that we developed to visualize how rules fire at different places in analyzed code.

STYLE-ANALYZER is based on LOOKOUT, a featureful framework for assisted code review. LOOKOUT allowed us to focus on more important tasks and develop code analyses faster. We hope that our framework will inspire other researchers to conduct assisted code review studies in the future. We release STYLE-ANALYZER as a reusable and extensible software package together with the datasets used for evaluation. We plan to turn STYLE-ANALYZER into a commercial product and improve the suggestions quality by analyzing how real users interact with the tool.

VIII. Acknowledgment

We warmly thank source{d} Applications Team members, namely Maxim Sukharev, Carlos Martin, Alexander Bezzubov, David Pordomingo and Lou Marvin Caraig, for their hard work and diligent support on the LOOKOUT framework.

References

[1] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, pp. 861–867, Nov. 1983.

[2] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: Indentation as a proxy for complexity metric," in *Proceedings of the 16th International Conference on Program Comprehension*, ser. ICPC '08. IEEE, 2008, pp. 133–142.

[3] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, Jul. 2009.

[4] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. ACM, 2018, pp. 181–190.

[5] M. Brand, van den, A. Kooiker, J. Vinju, and N. Veerman, "A language independent framework for context-sensitive formatting," in *Proceedings 10th European Conference on Software Maintenance and Reengineering*, ser. CSMR '06. IEEE, 2006, pp. 103–112.

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, ser. FSE '14. ACM, 2014, pp. 281–293.

[7] T. Parr and J. Vinju, "Towards a universal code formatter through machine learning," in *Proceedings of the 9th International Conference on Software Language Engineering*, ser. SLE '16. ACM, 2016, pp. 137–151.

[8] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proceedings of the 18th Working Conference on Reverse Engineering*, ser. WCRE '11. IEEE, 2011, pp. 35–44.

[9] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. FSE '13. ACM, 2013, pp. 532–542.

[10] "Babelfish," https://github.com/bblfsh, visited January 15, 2019.

[11] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. ACM, 2018, pp. 542–553.

[12] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning models that remember too much," in *Proceedings of the Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 587–601.

[13] J. Tang, S. Alelyani, and H. Liu, "Feature selection for classification: A review," *Data classification: Algorithms and applications*, p. 37, 2014.

[14] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," http://www.scipy.org/, 2001–, visited January 15, 2019.

[15] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[16] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[17] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] T. Head, MechCoder, G. Louppe, I. Shcherbatyi, fcharras, Z. Vinícius, cmmalone, C. Schröder, nel215, N. Campos, T. Young, S. Cereda, T. Fan, rene rex, K. K. Shi, J. Schwabedal, carlosdanielcsantos, Hvass-Labs, M. Pak, SoManyUsernamesTaken, F. Callaway, L. Estève, L. Besson, M. Cherti, K. Pfannschmidt, F. Linzberger, C. Cauet, A. Gut, A. Mueller, and A. Fabisch, "scikit-optimize," 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1207017

[19] J. R. Quinlan, "Generating production rules from decision trees," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, ser. IJCAI '87. Morgan Kaufmann Publishers Inc., 1987, pp. 304–307.

[20] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[21] P. Greenfield, M. Droettboom, and E. Bray, "ASDF: A new data format for astronomy," *Astronomy and Computing*, vol. 12, pp. 240–251, Sep. 2015.

[22] R. Vilalta and Y. Drissi, "A perspective view and survey of meta-learning," *Artificial Intelligence Review*, vol. 18, no. 2, pp. 77–95, 2002.

[23] A. A. Rusu, D. Rao, J. Sygnowski, O. Vinyals, R. Pascanu, S. Osindero, and R. Hadsell, "Meta-learning with latent embedding optimization," in *Proceedings of the 2nd NeurIPS Workshop on Meta-Learning*, ser. MetaLearn '18, 2018.