technologies

MDPI

Article

Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics

Gábor Antal [1], Zoltán Tóth [1], Péter Hegedűs [1,2,*] and Rudolf Ferenc [1]

[1] Department of Software Engineering, University of Szeged, Dugonics tér 13., 6720 Szeged, Hungary; antal@inf.u-szeged.hu (G.A.); zizo@inf.u-szeged.hu (Z.T.); ferenc@inf.u-szeged.hu (R.F.)
[2] MTA-SZTE Research Group on Artificial Intelligence, Dugonics tér 13., 6720 Szeged, Hungary
* Correspondence: hpeter@inf.u-szeged.hu

Abstract: Bug prediction aims at finding source code elements in a software system that are likely to contain defects. Being aware of the most error-prone parts of the program, one can efficiently allocate the limited amount of testing and code review resources. Therefore, bug prediction can support software maintenance and evolution to a great extent. In this paper, we propose a function level JavaScript bug prediction model based on static source code metrics with the addition of a hybrid (static and dynamic) code analysis based metric of the number of incoming and outgoing function calls (HNII and HNOI). Our motivation for this is that JavaScript is a highly dynamic scripting language for which static code analysis might be very imprecise; therefore, using a purely static source code features for bug prediction might not be enough. Based on a study where we extracted 824 buggy and 1943 non-buggy functions from the publicly available BugsJS dataset for the ESLint JavaScript project, we can confirm the positive impact of hybrid code metrics on the prediction performance of the ML models. Depending on the ML algorithm, applied hyper-parameters, and target measures we consider, hybrid invocation metrics bring a 2–10% increase in model performances (i.e., precision, recall, F-measure). Interestingly, replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improves model performances; however, using them all together yields the best results.

Keywords: bug prediction; hybrid code analysis; call-graph; source code metrics

Citation: Antal, G.; Tóth, Z.; Hegedűs, P.; Ferenc, R. Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics. *Technologies* **2021**, *9*, 3. http://doi.org/10.3390/technologies9010003

Received: 22 November 2020
Accepted: 23 December 2020
Published: 30 December 2020

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Bug prediction aims at finding source code elements in a software system that are likely to contain defects. Being aware of the most error-prone parts of the program, one can efficiently allocate the limited amount of testing and code review resources. Therefore, bug prediction can support software maintenance and evolution to a great extent. However, practical adoption of such prediction models always depends on their real-world performance and the level of disturbing misclassification (i.e., false-positive hits) they produce. Despite the relative maturity of the bug prediction research area, the practical utilization of the state-of-the-art models is still very low due to the reasons mentioned above.

Bug prediction models can use a diverse set of features to build effective prediction models. The most common types of such features are static source code metrics [1–4], process metrics [5–7], natural language features [8,9], and their combination [10–12]. All these metrics proved to be useful in different contexts, but the performance of these models may vary based on, for example, the language of the project, the composition of the project team, or the domain of the software product. We need further studies to understand better how and when these models work best in certain situations. Additionally, we can refine source code metrics by using static and dynamic analysis in combination, which has a yet unknown impact on the performance of bug prediction models.

In this paper, we propose a function level JavaScript bug prediction model based on static source code metrics with the addition of a hybrid (static and dynamic) code

analysis based metrics of the number of incoming and outgoing function calls. JavaScript is a highly dynamic scripting language for which static code analysis might be very imprecise. Although static source code metrics proved to be very efficient in bug prediction, the imprecision due to the lack of dynamic information might affect the bug prediction models using on them.

To support the hybrid code analysis of JavaScript programs, we created a hybrid call-graph framework, called hcg-js-framework (https://github.com/sed-szeged/hcg-js-framework) that can extract call-graph information of JavaScript functions using both static and dynamic analysis. Based on the hybrid call-graph results of the ESLint (https://eslint.org/) JavaScript project, which we used as a subject system for bug prediction, we refined the Number of Incoming Invocations (NII) and Number of Outgoing Invocations (NOI) metrics. We added them to a set of common static source code metrics to form the predictor features in a training dataset consisting of 824 buggy and 1943 non-buggy functions extracted from the publicly available BugsJS [13] bug dataset (https://github.com/BugsJS). These invocation metrics are typically very imprecise in JavaScript calculated based only on static analysis, as lots of calls happen dynamically, like higher-order function calls, changes in prototypes, or executing the *eval()* function, which is impossible to capture statically. We analyzed the impact of these additional hybrid source code metrics on the function-level bug prediction models trained on this dataset.

We found that using invocation metrics calculated by a hybrid code analysis as bug prediction features consistently improves the performance of the ML prediction models. Depending on the ML algorithm, applied hyper-parameters, and target measure we consider, hybrid invocation metrics bring a 2–10% increase in model performances (i.e., precision, recall, F-measure). Interestingly, even though replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improve model performances, keeping them all together yields the best results. It implicates that hybrid call metrics indeed add some complementary information to bug prediction.

The rest of the paper is structured as follows. In Section 2 we overview the JavaScript call-graph related literature and their usage for refining static source code metrics. We summarize our methodology for collecting ESLint bugs, mapping them to functions, extracting hybrid call-graphs, and assembling the training dataset in Section 3. Section 4 contains the results of comparing bug prediction models using only static, only hybrid, or both static and hybrid metrics as features for machine learning models. We enlist the possible threats to our work in Section 5 and conclude the paper in Section 6.

2. Related Work

Using call-graphs for source code and program analysis is a well-established and mature technique; the first papers dealing with call-graphs date back to the 1970's [14–16]. Call-graphs can be divided into two subgroups based on the method used to construct them: dynamic [17] and static [18].

Dynamic call-graphs can be obtained by the actual run of the program. During the run, several runtime information is collected about the interprocedural flow [19]. Techniques such as instrumenting the source code can be used for dynamic call-graph creation [16,20].

In contrast, there is no need to run the program in the case of static call-graphs, as it is produced by a static analyzer which analyzes the source code of software without actually running it [16]. On the other hand, static call-graphs might include false edges (calls) since a static analyzer identifies several possible calls between functions that are not feasible in the actual run of a program; or they might miss real edges. Static call-graphs can be constructed in almost any case from the source code, even if the code itself is not runnable.

Different analysis techniques are often combined to obtain a hybrid solution, which guarantees a more precise call-graph, thus a more precise analysis [21].

With the spread of scripting languages such as Python and JavaScript, the need for analyzing programs written in these languages also increased [22]. However, constructing precise static call-graphs for dynamic scripting languages is a very hard task that is not

fully solved yet [23]. The *eval()*, *apply()*, and *bind()* constructions of the languages make it especially hard to analyze the code statically. There are several approaches to construct such static call-graphs for JavaScript with varying success [22,24–26]. However, the most reliable method is to use dynamic approaches to detect such call edges. We decided to use both dynamic and static analysis to ensure better precision even though it increases the analysis time, and the code should be in a runnable state due to the dynamic analysis.

Wei and Ryder presented blended taint analysis for JavaScript, which uses a combined static-dynamic analysis [27]. By applying dynamic analysis, they could collect information for even those situations that are hard to analyze statically. Dynamic results (execution traces) are propagated to a static infrastructure, which embeds a call-graph builder as well. This call-graph builder module makes use of the dynamically identified calls. However, in the case of pure static analysis, they wrapped the WALA tool (https://github.com/wala/WALA) to construct a static call-graph. As previously said, our approach works similarly and also supports additional call graph builder tools to be included in the flow of the analysis.

Feldthaus et al. presented an approximation method to construct a call-graph [22] by which a scalable JavaScript IDE support could be guaranteed. We used a static call-graph builder tool in our toolchain, which is based on this approximation method (https://github.com/Persper/js-callgraph). Additional static JavaScript call-graph building algorithms were evaluated by Dijkstra [28]. Madsen et al. focused on the problems induced by libraries used in the project [29]. They used pointer analysis and a novel "use analysis" to enhance scalability and precision.

There are also works intending to create a framework for comparing call-graph construction algorithms [30,31]. However, these are done for algorithms written in Java and C. Call-graphs are often used for preliminary analysis to determine whether an optimization can be done on the code or not. Unfortunately, as they are specific to Java and C, we could not use these frameworks for our paper.

Clustering call-graphs can have advantages in malware classification [32], they can help localizing software faults [33], not to mention the usefulness of call-graphs in debugging [34].

Musco et al. [35] used four types of call-graphs to predict the software elements that are likely to be impacted by a change in the software. However, they used mutation testing to assess the impact of a change in the source code. The same methodology could have been used but with a slight change: instead of using an arbitrary change, it can be a vulnerability introducing or a vulnerability mitigating change.

Nuthan Munaiah and Andrew Meneely [36] introduced two novel attack surface metrics with their approach, which are the "Proximity" and "Risky Walks" metrics. Both of them are defined by the call-graph representation of the program. Their empirical study proved that using their metrics to build a prediction model can help to predict more accurately as their metrics are statistically significantly associated with the vulnerable functions.

Nguyen et al. [37] proposed a model to predict vulnerable components based on a metric set generated from the component dependency graph of a software.

Cheng et al. [38] presented a new approach to detect control-flow-related vulnerabilities called VGDetector. They applied a recent graph convolutional network to embed code fragments in a compact representation (while the representation still preserves the high-level control-flow information).

Neuhaus et al. [39] presented a fully automatic way to map vulnerabilities to software components and a tool called Vulture that can automatically build predictors to predict vulnerabilities in a new component. They identified that imports and function calls have an impact on whether a component vulnerable or not. They also made an evaluation of Mozilla's codebase that showed that their approach is accurate.

Lee et al. [40] proposed a new approach to generate semantic signatures from programs to detect malware. They extracted the call-graph of the API call sequence that would be generated by malware, called code graph. This graph is used for the semantic signature.

They used semantic signatures to detect malware even if the malware is obfuscated or the malware slightly differs from its previous versions (these are the main reasons why a commercial anti-virus does not detect them as malware).

As these previous studies show, the advantage of call-graphs is present in predicting vulnerabilities in software systems. We did not narrow down the type of defects. Our approach is generally applicable to arbitrary bug prediction.

The most similar to our study is possibly the work of Punia et al. [41] who presented a call-graph based approach to predict and detect defects in a given program. They also defined call-graph based metrics such as Fan In, Fan Out, Call-Graph Based Ranking (CGBR) and Information Flow Complexity (IFC). They investigated the correlation between their metrics and several types of bugs. They proved the hypothesis that there is a correlation between call-graph based metrics and bugs in software design. The authors performed their study in the Java domain; contrarily, we focused on JavaScript systems. Besides J84, LMT, and SMO, we applied additional machine learning algorithms and also evaluated deep learning techniques to find potential bugs in the software. As many papers, we also focused on different source code metrics; however, we adopted coupling metrics for the so-called hybrid call-graph.

3. Methodology

Our approach consists of numerous steps, which we present in detail in this section. Figure 1 shows the steps required to produce input for the machine learning algorithms.

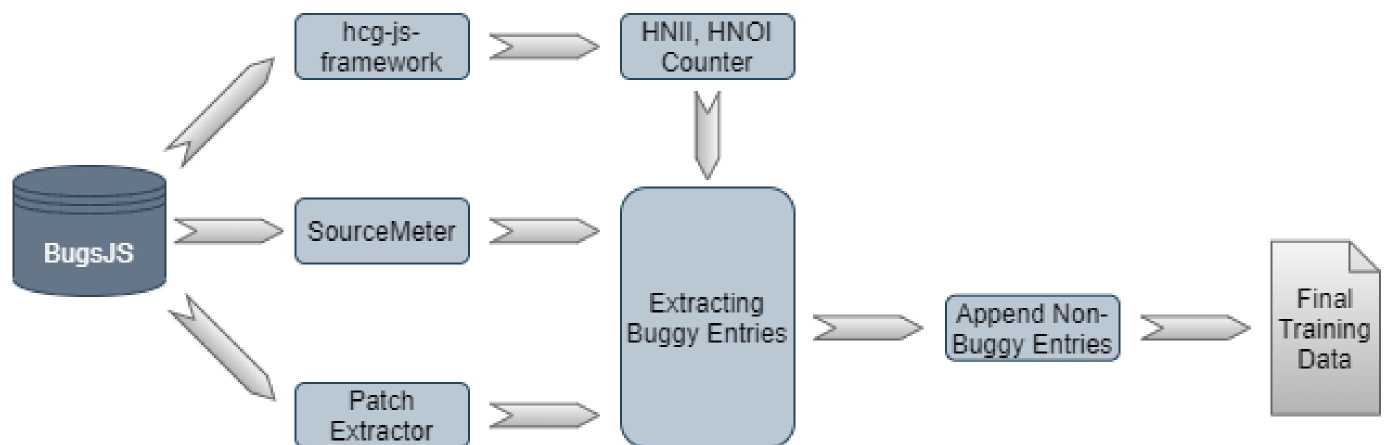

Figure 1. Applied Toolchain.

3.1. BugsJS Dataset

BugsJS [13] is a bug dataset inspired by Defects4J [42]; however, it provides bug related information for popular JavaScript-based projects instead of Java projects. Currently, BugsJS includes bug information for ten projects that are actively maintained Node.js server-side programs hosted on GitHub. Most importantly, BugsJS includes projects which adopt the Mocha testing framework; consequently, we can implement dynamic analysis experiments easier.

BugsJS stores the forks of the original repositories and extends them by adding tags for their custom commits in the form of:

- `Bug-X`: The parent commit of the revision in which the bug was fixed (i.e., the buggy revision)
- `Bug-X-fix`: A revision (commit) containing only the production code changes (test code and documentation changes were excluded) introduced in order to fix the bug

where `X` denotes a number associated with a given bug. As out of the total 453 bugs, ESLint (https://github.com/eslint/eslint) itself contains 333 bugs, we chose this project as input in our study.

3.2. Hybrid Invocation Metrics Calculation

As a first step, we have to produce the so-called hybrid call-graphs from which we can calculate the hybrid invocation metrics (i.e., HNII and HNOI). In order to understand what a hybrid call-graph is, let us consider Figure 2, which shows the details of the node "hcg-js-framework" presented earlier in Figure 1.

As can be seen, the input of the hcg-js-framework is the JavaScript source code that we want to analyze, which can be either a Git repository or a local folder. Then we analyze the source code with various static and dynamic tools. Following the analyses, the framework converts all the tool-specific outputs to a unified JSON format. Once we have the JSON files, the framework combines them into a merged JSON. This merged JSON contains every node and edge (JavaScript function nodes and call edges between them) that either of the tools found.

After this step, we augment this merged JSON with confidence levels for the edges. The confidence levels are calculated based on a manual evaluation of 600 out of 82,791 call edges found in 12 real-world Node.js modules. We calculated the True Positive Rate for each tool intersection. We estimate the confidence of a call edge with these rates. For instance, if a call edge was found by tools A and B, and in the manually evaluated sample, there were ten edges found by only these tools, from which five turned out to be a valid call edge, we add confidence of 0.5 to all these edges.

Figure 3 shows a Venn diagram of the call edges found in 12 Node.js modules. We have an evaluation ratio for each intersection, which the framework uses for edge confidence level estimation.

To sum it up, a hybrid call-graph is a call-graph (produced by combining the results of both static and dynamic analysis) which associates a confidence factor to each call edge, which shows how likely an edge is valid (higher confidence means higher validity).

This hybrid call-graph is the input of the HNII, HNOI Counter which is responsible for calculating the exact number of incoming and outgoing invocations (i.e., NII, NOI). At this point, we have to specify the threshold value, which defines the lower limit from which we consider a call edge as a valid call edge, thus contributing to the value of the number of incoming and outgoing invocations. We considered four threshold values: 0.00, 0.05, 0.20 and 0.30. In the case of the first one, all edges are considered as possibly valid call edges, while the latter one only includes edges with a high confidence factor. We name these two new metrics as HNII (Hybrid Number of Incoming Invocations) and HNOI (Hybrid Number of Outgoing Invocations) to differentiate them from the original static NII and NOI metrics.

The HNII, HNOI Counter traverses all call edges and considers those edges where the confidence level is above the given threshold. The edges fulfilling this threshold criteria contribute to the HNOI metric of the source node and the HNII metric of the target node. As a result, the tool produces a JSON file as its output, which contains only the nodes (i.e., the JavaScript functions) with their corresponding HNII, HNOI metric values, and additional information about their position in the system, such as source file, line, and column. Listing 1 shows an example of a single node output.

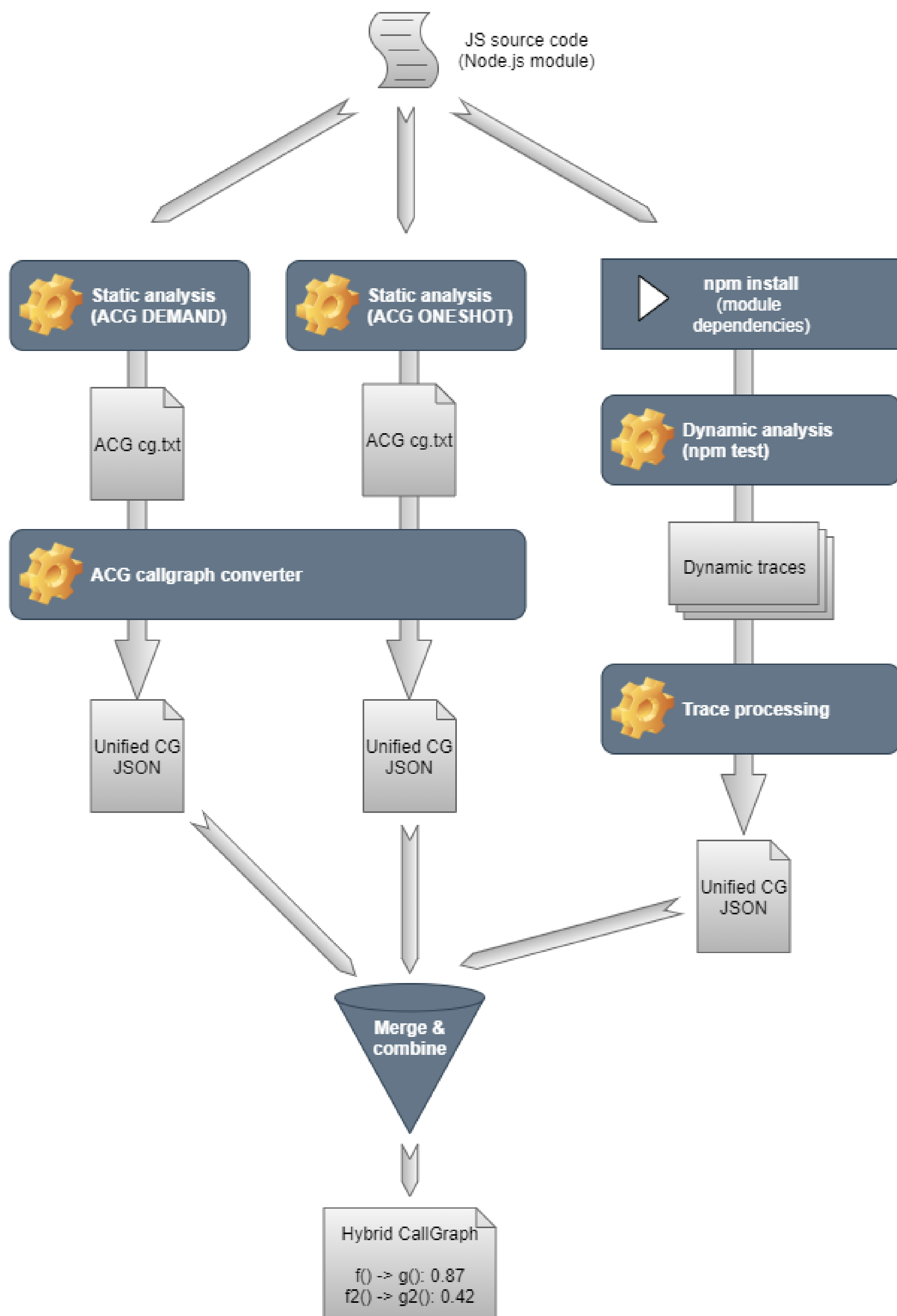

Figure 2. Hybrid call-graph framework architecture.

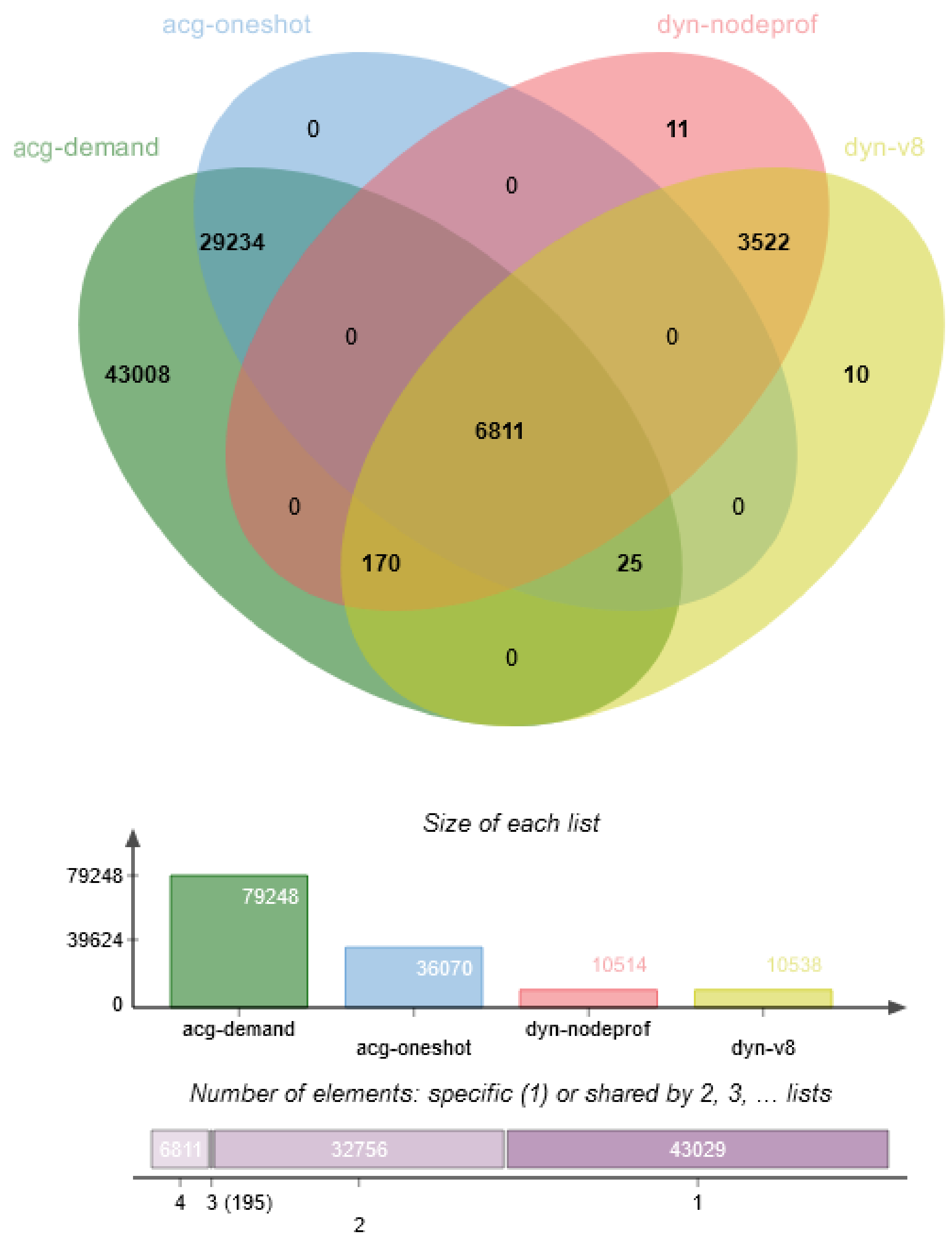

Figure 3. Venn diagram of found edges.

Listing 1: Sample output from the HNII, HNOI Counter.

```
{
"pos":"eslint/lib/ast-utils.js:169:25",
"entry":false,
"final":false,
"hnii":1,
"hnoi":3
}
```

3.3. SourceMeter and Patch Extraction

Besides computing the HNII and HNOI metrics, a standard set of metrics is provided by a static source code analyzer named SourceMeter (https://www.sourcemeter.com/). SourceMeter also takes JavaScript source code as input and outputs (amongst others) different CSV files for different source code elements (functions, methods, classes, files, system). In this study, we used the resulting CSV file that contains function level entries, which captures static size metrics (LOC, LLOC, NOS), complexity metrics (McCC, NL), documentation metrics (CD, CLOC, DLOC), and traditional coupling metrics (NII, NOI) as well. These metrics are calculated for all the 333 bugs in ESLint before and after the bug is fixed, which means 666 static analyses in total.

Similarly, we extracted the patches for these 333 bug fixing commits, which is done by Patch Extractor.

3.4. Composing Buggy Entries

At this point, we have all the necessary inputs to combine them in one CSV, which contains the buggy entries with their static source code metrics extended with the HNII and HNOI metrics. The core of the algorithm is the following. We traverse all the bugs one-by-one. For bug_i , we retrieved a set of entries from the ith static analysis results, which were touched by the fixing $patch_i$ (determined based on entry name and positional information) and extended these entries with the corresponding HNII and HNOI metric values. We included the before-fix state (i.e., the buggy) for the touched JavaScript functions, and used the date of the latest bug to select non-buggy instances from that corresponding version of the code (i.e., Bug-79 fixed at 2018-03-21 17:23:34). For non-buggy entries, we also extracted the corresponding HNII and HNOI values also from the latest buggy version.

4. Results

To calculate the HNOI and HNII metrics, one needs to apply a threshold to the call edges (to decide which edges to consider as valid) in the underlying hybrid (also called as fuzzy) call-graph produced by the hcg-js-framework (see Section 3). We calculated the metric values (all the data used in this study is available online [43]) with four different thresholds: 0, 0.05, 0.2, and 0.3. Table 1 shows the descriptive statistics of the metrics on our ESLint dataset.

Table 1. Descriptive statistics of the HNII and HNOI (Hybrid Number of Incoming/Outgoing Invocations) metrics calculated using different thresholds.

	HNII			HNOI		
Threshold	**Avg.**	**Median**	**Std.dev.**	**Avg.**	**Median**	**Std.dev.**
0.00	7.026021	1	26.95583	5.341887	2	27.27586
0.05	6.96133	1	26.95997	5.243224	2	26.91228
0.20	0.840622	1	2.823739	1.018793	0	9.236607
0.30	0.840622	1	2.823739	1.018793	0	9.236607

As can be seen, thresholds 0.20 and above significantly reduces the number of considered edges for HNII and HNOI calculation. We wanted to use as many of the extracted call edges as possible, so we selected to use the 0.00 threshold later on (i.e., we considered each edge in the fuzzy call-graph where the weight/confidence is greater or equal to zero).

We trained several models on the dataset with three different configurations for the features:

- Purely static metrics ($S - 0_00_s.csv$): the dataset contains only the pure static source code metrics (i.e., original versions of NOI and NII plus all the provided metrics by SourceMeter, see Section 3.3).

- Static metrics with only hybrid NOI and NII versions ($H - 0_00_h.csv$): the dataset contains all the static metrics except NOI and NII, which are replaced by their hybrid counterparts (HNOII and HNII) calculated on the output of *hcg-js-framework*.
- Both static and hybrid metrics ($S + H - 0_00_s + h.csv$): the dataset contains all the static metrics plus the hybrid counterparts of NOI and NII (HNOII and HNII) calculated on the output of *hcg-js-framework*.

To have a robust understanding of the hybrid metrics' impact, we trained nine different machine learning models:

- Logistic Regression Classifier—Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable (implemented by `sklearn.linear_model.LogisticRegression`);
- Naive Bayes Classifier—Naive Bayes classifier is a simple "probabilistic classifier" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features (implemented by `sklearn.naive_bayes.GaussianNB`);
- Decision Tree Classifier—Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression, where the goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features (implemented by `sklearn.tree.DecisionTreeClassifier` an optimized version of the CART algorithm);
- Linear Regression Classifier—Linear regression is a linear approach to modeling the relationship between a scalar response and one or more explanatory variables also known as dependent and independent variables (implemented by `sklearn.linear_model.LinearRegression`);
- Standard DNN Classifier—A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers (implemented using `tensorflowf.layers.dense`);
- Customized DNN Classifier—A custom version of the standard DNN implementing th early stopping mechanism, where we do not train the models for a fixed number of epochs, rather stop when there is no more reduction in the loss function (implemented using `tensorflowf.layers.dense`);
- Support Vector Machine Classifier—Support-vector machine (SVM) is a supervised learning model, which is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible (implemented by `sklearn.svm.SVC`);
- K Nearest Neighbors Classifier—The k-nearest neighbors algorithm (k-NN) is a non-parametric method for classification and regression, where the input consists of the k closest training examples in the feature space (implemented by `sklearn.neighbors.KNeighborsClassifier`);
- Random Forest Classifier—Random forest is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees (implemented by `sklearn.ensemble.RandomForestClassifier`).

With the various hyper-parameters, it added up to a total of 36 configurations. We executed all these 36 training tasks on all three feature sets, so we created 108 different ML models for comparison. To cope with the highly imbalanced nature of the dataset (i.e., there are significantly more non-buggy functions than buggy ones), we applied a 50% oversampling on the minority class. We also standardized all the metric values to bring them to the same scale. For the model training and evaluation, we used our open-source DeepWater Framework (https://github.com/sed-inf-u-szeged/DeepWaterFramework) [44], which contains the implementation of all the above algorithms.

To ensure that the results are robust against the chosen threshold, we trained the same 108 models with the threshold value of 0.30 for HNII and HNOI calculation. We found that the differences among S, H, and $S + H$ feature sets became less, but the general tendency

that H and especially $S + H$ features achieved better results remained. Therefore, in the rest of the paper, we can use the HNII and HNOI metrics calculated with the 0.00 threshold without the loss of generality. In the remaining, we present our findings.

4.1. The Best Performing Algorithms

Figure 4 displays a heat-mapped table of the top 10 model results based on their recall values. We ranked all 108 models, meaning that all three feature sets are on the same list. We can measure recall with the following formula:

$$Recall = \frac{TP}{TP + FN},$$

where TP means True Positive samples, while FN means False Negatives. As we can see, DNN (0.642) and KNN (0.635) models achieve the best recall values on the $S + H$ feature set. The same models produce almost as high recall values (0.631) using only the H feature set. The best performing model on the S feature set is KNN, with a significantly lower (0.619) recall value. It shows that hybrid invocation metrics do increase the performance of ML models in terms of recall. The best values are achieved by keeping both the original NOI and NII metrics and adding their hybrid counterparts HNOI and HNII, but using only the latter ones as substitutes for the static metrics still improves recall values.

algorithm	file_path	test-accuracy	test-precision	test-recall ↓	test-fmes	test-mcc
Customized DNN Classifier	/SeTiT/0_00_s+h.csv	0.763 ± 0.031	0.595 ± 0.056	0.642 ± 0.074	0.618 ± 0.043	0.447 ± 0.063
K Nearest Neighbors Classifier	/SeTiT/0_00_s+h.csv	0.788 + 0.025	0.646 + 0.051	0.635 − 0.007	0.641 + 0.023	0.490 + 0.043
K Nearest Neighbors Classifier	/SeTiT/0_00_h.csv	0.776 + 0.012	0.621 + 0.026	0.631 − 0.011	0.626 + 0.008	0.466 + 0.019
Customized DNN Classifier	/SeTiT/0_00_h.csv	0.749 − 0.014	0.571 − 0.024	0.631 − 0.011	0.600 − 0.018	0.419 − 0.028
K Nearest Neighbors Classifier	/SeTiT/0_00_s+h.csv	0.772 + 0.009	0.617 + 0.022	0.619 − 0.023	0.618 + 0.000	0.455 + 0.008
K Nearest Neighbors Classifier	/SeTiT/0_00_s.csv	0.763 + 0.000	0.599 + 0.004	0.619 − 0.023	0.609 − 0.009	0.439 − 0.008
K Nearest Neighbors Classifier	/SeTiT/0_00_s+h.csv	0.787 + 0.024	0.650 + 0.055	0.619 − 0.023	0.634 + 0.016	0.484 + 0.037
K Nearest Neighbors Classifier	/SeTiT/0_00_s.csv	0.769 + 0.006	0.613 + 0.018	0.614 − 0.028	0.613 − 0.004	0.449 + 0.002
K Nearest Neighbors Classifier	/SeTiT/0_00_h.csv	0.778 + 0.014	0.630 + 0.035	0.613 − 0.029	0.622 + 0.004	0.464 + 0.017
K Nearest Neighbors Classifier	/SeTiT/0_00_h.csv	0.768 + 0.004	0.610 + 0.015	0.609 − 0.033	0.610 − 0.008	0.444 − 0.003

Figure 4. Top 10 recall measures.

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 5 and 6) with the best DNN configurations (i.e., applying the set of hyper-parameters with which the model achieves the best performance) for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S + H$ results are superior, while H results are better than the S results except for the False Positive and True Negative instances. The chart shows that there is a constant

3–4% improvement in all aspects of the DNN model results if we add the hybrid metrics to the feature sets.

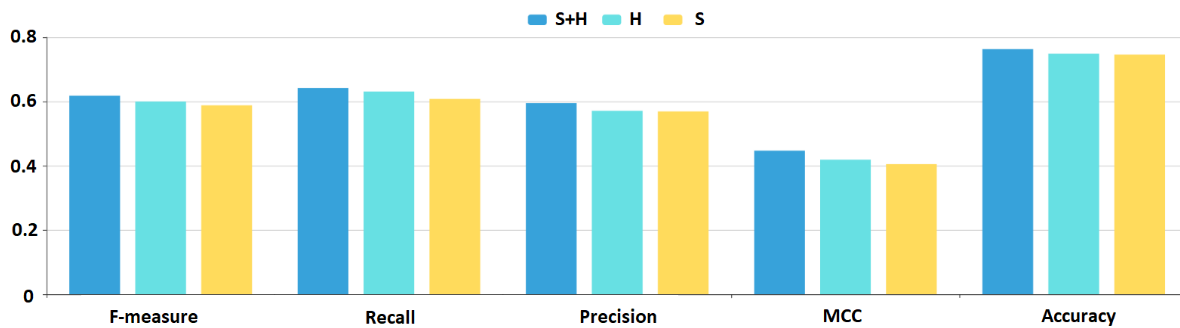

Figure 5. Deep neural network.

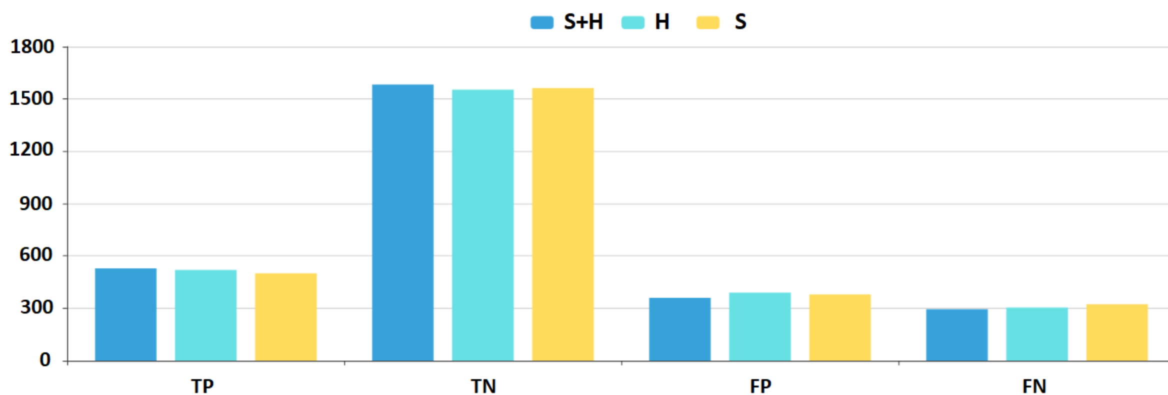

Figure 6. Deep neural network.

Figure 7 displays a heat-mapped table of the top 10 model results based on their precision values. We ranked all 108 models, meaning that all three feature sets are on the same list. We can measure precision with the following formula:

$$Precision = \frac{TP}{TP + FP},$$

where TP means True Positive samples, while FP means False Positives. As we can see, the SVM model (0.829) achieves the best precision values on the H feature set. Interestingly, SVM produces an almost as high precision value (0.827) using only the S feature set as well. Based on the $S + H$ feature set, SVM achieves a precision value of 0.824. It shows that hybrid invocation metrics do increase the performance of ML models in terms of precision, but not as significantly as in the case of recall values. Nonetheless, for other algorithms than SVM, the increase is more significant.

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 8 and 9) with the best SVM configurations for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S + H$ results are superior, while H results are still better than S results for all measures. The chart shows that there is a constant 1–2% improvement in all aspects of the SVM model results if we add the hybrid metrics to the feature sets.

algorithm	file_path	test-accuracy	test-precision ↓	test-recall	test-fmes	test-mcc
Support Vector Machine Classifier	/SeTiT /0_00_h.csv	0.756 ± 0.019	0.829 ± 0.069	0.229 ± 0.055	0.359 ± 0.073	0.348 ± 0.069
Support Vector Machine Classifier	/SeTiT /0_00_s.csv	0.757 + 0.000	0.827 − 0.002	0.232 + 0.002	0.362 + 0.003	0.349 + 0.001
Support Vector Machine Classifier	/SeTiT /0_00_s+h.csv	0.759 + 0.003	0.824 − 0.005	0.244 + 0.015	0.376 + 0.017	0.358 + 0.010
Support Vector Machine Classifier	/SeTiT /0_00_s.csv	0.759 + 0.003	0.767 − 0.062	0.275 + 0.046	0.405 + 0.046	0.355 + 0.007
Support Vector Machine Classifier	/SeTiT /0_00_h.csv	0.755 − 0.001	0.762 − 0.067	0.260 + 0.030	0.387 + 0.028	0.341 − 0.007
Support Vector Machine Classifier	/SeTiT /0_00_s+h.csv	0.760 + 0.004	0.756 − 0.073	0.289 + 0.059	0.418 + 0.059	0.359 + 0.011
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.816 + 0.060	0.753 − 0.076	0.569 + 0.340	0.648 + 0.289	0.536 + 0.188
Random Forest Classifier	/SeTiT /0_00_h.csv	0.814 + 0.057	0.743 − 0.086	0.573 + 0.343	0.647 + 0.288	0.532 + 0.184
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.808 + 0.052	0.731 − 0.098	0.564 + 0.335	0.637 + 0.278	0.518 + 0.170
Random Forest Classifier	/SeTiT /0_00_h.csv	0.810 + 0.053	0.726 − 0.103	0.579 + 0.350	0.644 + 0.285	0.523 + 0.174

Figure 7. Top 10 precision measures.

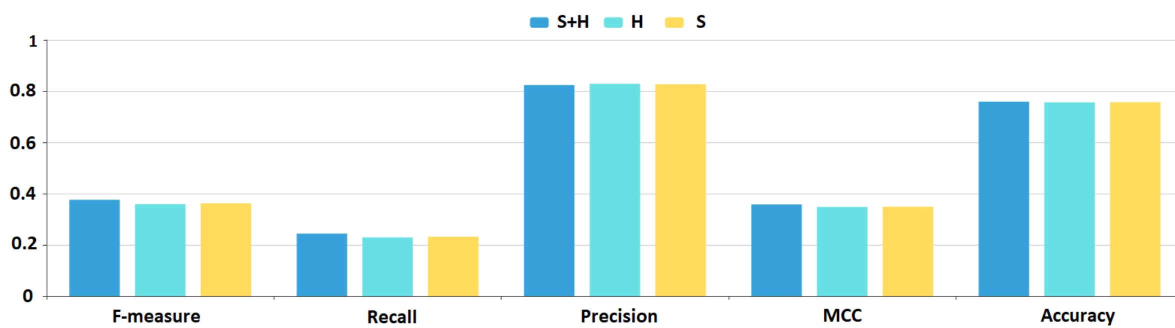

Figure 8. Support-vector machine (SVM).

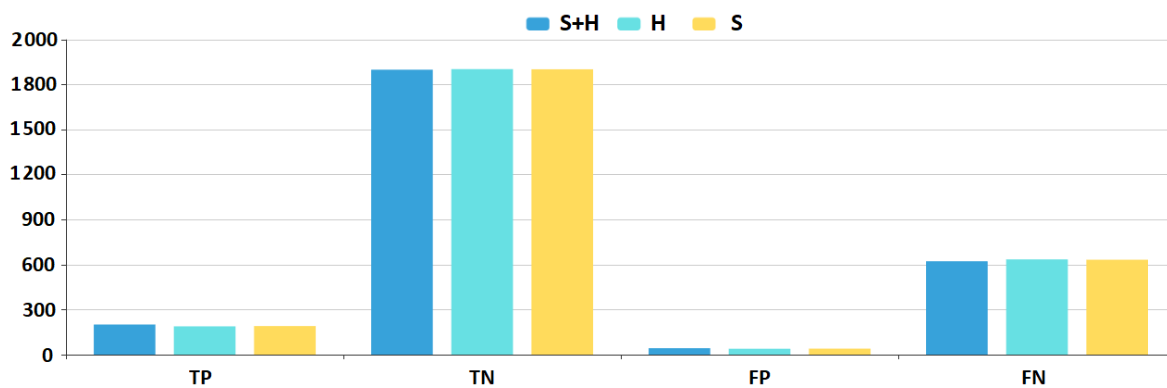

Figure 9. SVM.

Figure 10 displays a heat-mapped table of the top 10 model results based on their F-measure values. We ranked all 108 models, meaning that all three feature sets are on the same list. We can calculate F-measure with the following formula:

$$F - measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}.$$

As we can see, Random Forest (0.648) and KNN (0.641) models achieve the best F-measures on the $S + H$ feature set. The same Random Forest models produce almost as high F-measures (0.647) using only the H feature set. The best performing model on the S feature set is not even in the top 10. It shows that hybrid invocation metrics do increase the performance of ML models in terms of F-measure, meaning they improve the models' overall performance. The best values are achieved by keeping both the original NOI and NII metrics and adding their hybrid counterparts HNOI and HNII, but using only the latter ones as substitutes for the static metrics still improves F-measure significantly.

algorithm	file_path	test-accuracy	test-precision	test-recall	test-fmes ↓	test-mcc
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.816 ± 0.024	0.753 ± 0.046	0.569 ± 0.068	0.648 ± 0.054	0.536 ± 0.064
Random Forest Classifier	/SeTiT /0_00_h.csv	0.814 - 0.002	0.743 - 0.010	0.573 + 0.004	0.647 - 0.001	0.532 - 0.005
Random Forest Classifier	/SeTiT /0_00_h.csv	0.810 - 0.007	0.726 - 0.027	0.579 + 0.010	0.644 - 0.004	0.523 - 0.014
K Nearest Neighbors Classifier	/SeTiT /0_00_s+h.csv	0.788 - 0.028	0.646 - 0.106	0.635 + 0.066	0.641 - 0.008	0.490 - 0.046
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.805 - 0.011	0.712 - 0.041	0.579 + 0.010	0.639 - 0.010	0.512 - 0.024
Random Forest Classifier	/SeTiT /0_00_h.csv	0.799 - 0.017	0.690 - 0.063	0.592 + 0.023	0.637 - 0.011	0.503 - 0.034
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.808 - 0.008	0.731 - 0.022	0.564 - 0.005	0.637 - 0.011	0.518 - 0.019
Random Forest Classifier	/SeTiT /0_00_s+h.csv	0.799 - 0.017	0.690 - 0.062	0.587 + 0.018	0.635 - 0.013	0.500 - 0.036
K Nearest Neighbors Classifier	/SeTiT /0_00_s+h.csv	0.787 - 0.029	0.650 - 0.103	0.619 + 0.050	0.634 - 0.014	0.484 - 0.052
K Nearest Neighbors Classifier	/SeTiT /0_00_h.csv	0.776 - 0.040	0.621 - 0.132	0.631 + 0.062	0.626 - 0.022	0.466 - 0.071

Figure 10. Top 10 F-measures.

To visualize the difference in the various performance measures, we plotted a bar-chart (Figures 11 and 12) with the best Random Forest configurations for all three feature sets. Blue marks the results using the $S + H$ feature set, cyan the H feature set, while yellow the S feature set. $S + H$ and H results are better than S results for all measures except for recall, but the difference there is only marginal. The chart shows that there is a constant 1–2% improvement in all aspects of the Random Forest model results, but precision is higher by approximately 10% if we add the hybrid metrics to the feature sets.

Figure 11. Random forest.

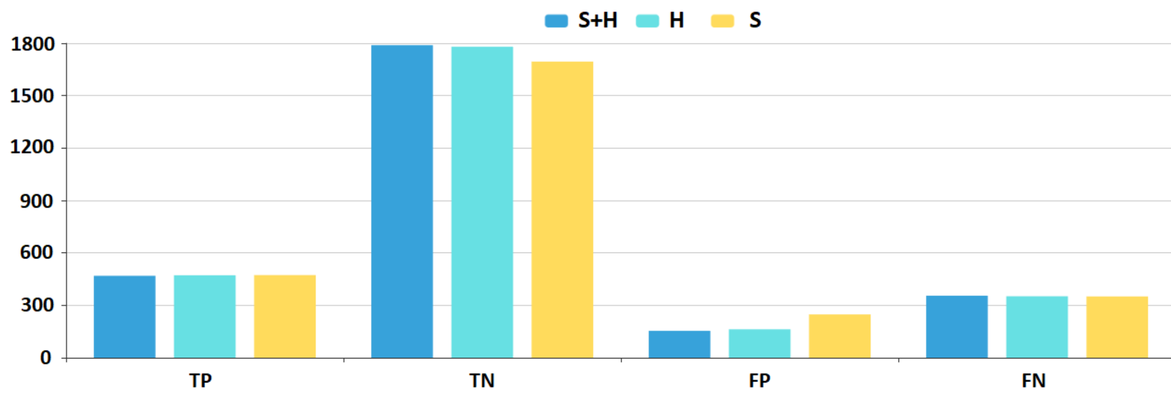

Figure 12. Random forest.

4.2. The Most Balanced Algorithm

K-nearest neighbor models stand out in that they produce the most balanced performance measures. As can bee sen in Figures 13 and 14, both precision and recall values are above 0.6, therefore F-measure is above 0.6 as well. For this model, H feature set brings a 1–2% improvement over the S feature set, while $S + H$ feature set results in a 2–5% increase in performance.

4.3. Significance Analysis of the Performance Measures

Despite a seemingly consistent increase in every model performance measures caused by adding hybrid source code metrics to the features, we cannot be sure that this improvement is statistically significant. Therefore, we performed a Wilcoxon signed-rank test [45] on the model F-measure values between each pair of feature sets (S vs. H, S vs. S + H, H vs. S + H). The detailed results (T statistics and *p*-values) are shown in Table 2.

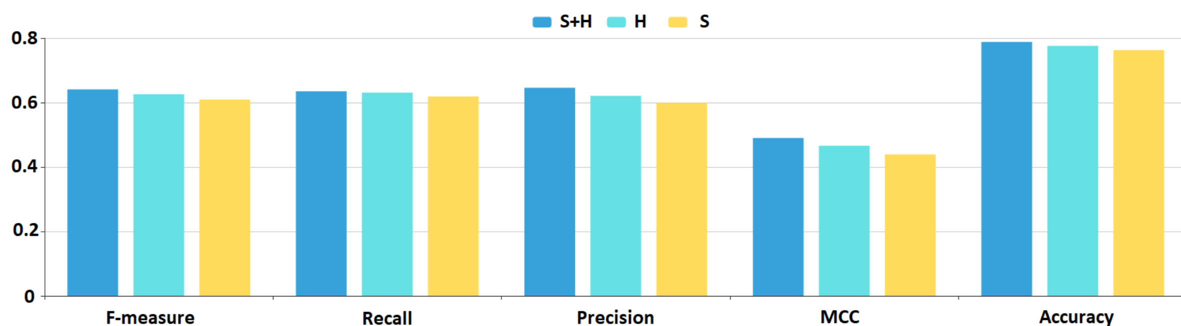

Figure 13. K-nearest neighbors.

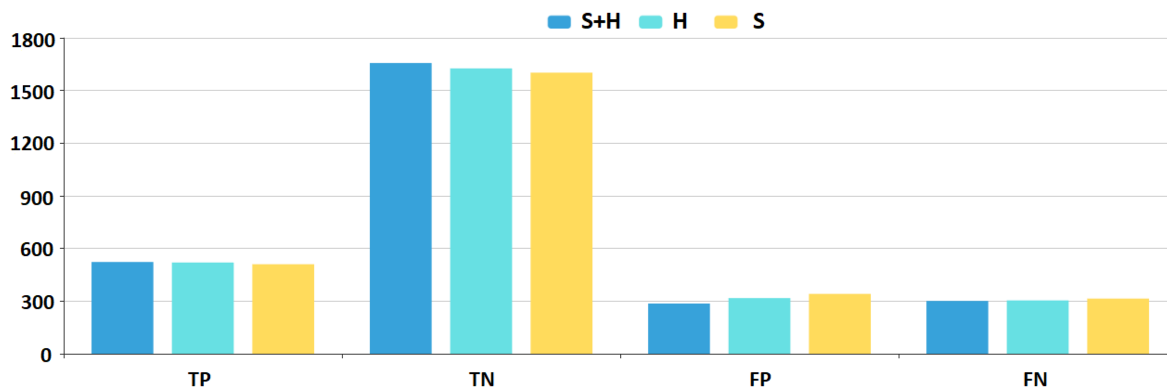

Figure 14. K-nearest neighbors.

Table 2. Wilcoxon signed-rank test results of F-measures between models using different feature sets.

	Hybrid		Static + Hybrid	
Features	**T**	***p*-Value**	**T**	***p*-Value**
>Static	95.5	0.00032	74	0.00004
>Hybrid	-	-	116	0.0019

As can be seen, the F-measure values achieved by the models differ significantly (*p*-value is less than 0.05) depending on the feature sets we used for training them. It is interesting to observe that there is a significant difference in performances even between the models using the hybrid and static+hybrid features and not just between models using static features only and models using hybrid features as well. These results confirm that even though hybrid source code metrics provide additional prediction power to bug prediction models, they do not substitute static source code metrics but complement them. Therefore, according to our empirical data, we could achieve the best performing JavaScript bug prediction models by keeping static call-related metrics and adding their hybrid counterparts to the model features. We note that the Wilcoxon signed-rank test showed significant results in the case of precision and recall performance measures as well.

4.4. Results Overview and Discussion

In the previous sections, we analyzed the best performing algorithms with a focus on the improvements caused by the hybrid source code metrics. However, to have a complete picture of the results, we summarize the performances of all nine machine learning algorithms here. Table 3 shows the best prediction performances (i.e., models with best performing hyper-parameters and feature set) of all nine algorithms according to their F-measures.

Table 3. The best results of the nine ML algorithms according to their F-measure.

ML Algorithm	Feature Set	Accuracy	Precision	Recall	F-Measure	MCC
>Random Forest Classifier	S + H	0.816	0.753	0.569	0.648	0.54
>K Nearest Neighbors Classifier	S + H	0.788	0.646	0.635	0.641	0.49
>Customized DNN Classifier	S + H	0.784	0.649	0.601	0.624	0.47
>Decision Tree Classifier	S + H	0.781	0.649	0.58	0.612	0.46
>Standard DNN Classifier	H	0.774	0.634	0.569	0.6	0.44
>Logistic Regression Classifier	S + H	0.787	0.682	0.533	0.598	0.46
>Support Vector Machine Classifier	S + H	0.789	0.699	0.515	0.593	0.47
>Linear Regression Classifier	S + H	0.769	0.67	0.443	0.533	0.4
>Naive Bayes Classifier	S + H	0.772	0.713	0.394	0.508	0.4

There are several properties to observe in the table. First, all but one ML model achieves the best results in terms of F-measure using the $S + H$ feature set. The only exception is the Standard DNN Classifier, which performs best using only the hybrid version of call metrics (i.e., H feature set). Second, the Random Forest Classifier has the best F-measure (0.648) but also the best accuracy (0.816), precision (0.753), and MCC (0.54) metrics, which mean it performs the best overall for predicting software bugs in the studied JavaScript program. However, there is a trade-off between precision and recall; therefore, the Random Forest Classifier has only the third-highest recall metric (569). Third, the K Nearest Neighbors Classifier has the one but last lowest precision (0.646) and only the third-highest accuracy (0.788), still its recall (0.635) is the highest among all the models and in terms of F-measure (0.641) and MCC (0.49) it is a very close second behind the Random Forest Classifier. It implicates that the K Nearest Neighbors Classifier achieves the most balanced performance, not the best in every aspect but very high performance measures with no large variance. Fourth, the deep neural networks (Standard DNN Classifier and Customized DNN Classifier) do not outperform the simpler, classical models in this prediction task. The most likely cause of this is the relatively small amount of training samples, so the real strength of deep learning cannot be exploited. Fifth, the models struggle to achieve high recall values in general, which seems to be the bottleneck of the maximum F-measures. Even the worst-performing models (Linear Regression Classifier and Naive Bayes Classifier) have an acceptable accuracy (0.769 and 0.772, respectively) and precision (0.67 and 0.713, respectively), but very low recall (0.443 and 0.394, respectively), which results in a very low F-measure (0.533 and 0.508, respectively).

To sum up our experiences, it is worthwhile to add hybrid call metrics to the set of standard static source code metrics for training a JavaScript bug prediction model. To achieve the highest accuracy and precision, one should choose the Random Forest Classifier method, but if the recall is also important and one wants to have as balanced results as possible, the K Nearest Neighbors Classifier is the best possible option.

5. Threats to Validity

There are several threats to the validity of the presented empirical study. As a training set, we used 333 bugs only from one system. Therefore, the results might be specific to this system and might not generalize well. However, ESLint is a large and diverse program containing a representative set of issues. Additionally, bugs are manually filtered, thus do not introduce noise in the prediction models. As a result, we believe that our study is meaningful, though replication with more subject systems would be beneficial.

The threshold value chosen for calculating the hybrid call edges might affect the ML model performances. We selected a threshold of 0 (i.e., counted every edge with a weight greater than zero) in our case study; however, we carried out a sensitivity analysis with different thresholds as well. Even though the calculated HNOI and HNII values changed based on the applied threshold, the model improvements using these values proved to be consistent with the ones presented in the study. Therefore, we believe that the essence of the results is independent of the choice of the particular threshold value.

Finally, the provided thresholds might be inaccurate as we derived them from a manual evaluation of a small sample of real call edge candidates. To eliminate the risk of human error, two senior researchers evaluated all the edges who had to agree on each call label. For sampling, we applied a stratified selection strategy, so we evaluated more call samples from subsets of tools finding more edges in general, thus increasing the confidence of the derived weights.

6. Conclusions

In this paper, we proposed a function level JavaScript bug prediction model based on static source code metrics with the addition of a hybrid (static and dynamic) code analysis based metrics for incoming and outgoing function calls. JavaScript is a highly dynamic

scripting language for which static code analysis might be very imprecise; therefore, combining static and dynamic analysis to extract features is a promising approach.

We created three versions of a training dataset from the functions of the ESLint project. We used the BugsJS public dataset to find, extract, and map buggy functions in ESLint. We ended up with a dataset containing 824 buggy and 1943 non-buggy functions with three sets of features: static metrics only, static metrics where the invocation metrics (NOI and NII) are replaced by their hybrid counterparts (HNOI and HNII), static metrics with the addition of the hybrid metrics.

We trained nine different models in 108 configurations and compared their results. We found that using invocation metrics calculated by a hybrid code analysis as bug prediction features consistently improves the performance of the ML prediction models. Depending on the ML algorithm, applied hyper-parameters, and target measure we consider, hybrid invocation metrics bring a 2–10% increase in model performances (i.e., precision, recall, F-measure). Interestingly, even though replacing static NOI and NII metrics with their hybrid counterparts HNOI and HNII in itself improves model performances, most of the time, keeping them both yields the best results. This means that they hold somewhat complementary information to each other. To achieve the highest accuracy and precision, one should choose the Random Forest Classifier method, but if the recall is also important and one wants to have as balanced results as possible, the K Nearest Neighbors Classifier is the best possible option.

Author Contributions: Data curation, Z.T.; Formal analysis, P.H.; Investigation, G.A.; Methodology, G.A., Z.T. and P.H.; Project administration, P.H.; Software, Z.T.; Visualization, P.H.; Writing—original draft, Z.T. and P.H.; Writing—review & editing, G.A. and R.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the SETIT Project (2018-1.2.1-NKP-2018-00004), and partially supported by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All research data is available for download at https://doi.org/10.5281/zenodo.4281476.

Acknowledgments: Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-20-5-SZTE-650 New National Excellence Program of the Ministry for Innovation and Technology.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gray, D.P.H. Software defect prediction using static code metrics: Formulating a methodology. *Univ. Herts. Res. Arch.* **2013**. [CrossRef]
2. Li, L.; Leung, H. Mining static code metrics for a robust prediction of software defect-proneness. In Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement, Banff, AB, Canada, 22–23 September 2011; pp. 207–214.
3. Gray, D.; Bowes, D.; Davey, N.; Sun, Y.; Christianson, B. Using the support vector machine as a classification method for software defect prediction with static code metrics. In Proceedings of the International Conference on Engineering Applications of Neural Networks, London, UK, 27–29 August 2009; pp. 223–234.
4. Ferzund, J.; Ahsan, S.N.; Wotawa, F. Analysing bug prediction capabilities of static code metrics in open source software. In *Software Process and Product Measurement*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 331–343.
5. Hata, H.; Mizuno, O.; Kikuno, T. Bug prediction based on fine-grained module histories. In Proceedings of the 2012 34th international conference on software engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 200–210.
6. Shivaji, S.; Whitehead, E.J.; Akella, R.; Kim, S. Reducing features to improve code change-based bug prediction. *IEEE Trans. Softw. Eng.* **2012**, *39*, 552–569. [CrossRef]
7. Madeyski, L.; Jureczko, M. Which process metrics can significantly improve defect prediction models? An empirical study. *Softw. Qual. J.* **2015**, *23*, 393–422. [CrossRef]

8. Binkley, D.; Feild, H.; Lawrie, D.; Pighin, M. Increasing diversity: Natural language measures for software fault prediction. *J. Syst. Softw.* **2009**, *82*, 1793–1803. [CrossRef]
9. Haiduc, S.; Arnaoudova, V.; Marcus, A.; Antoniol, G. The use of text retrieval and natural language processing in software engineering. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 22 May 2016; pp. 898–899.
10. Alshehri, Y.A.; Goseva-Popstojanova, K.; Dzielski, D.G.; Devine, T. Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them. In Proceedings of the SoutheastCon 2018, St. Petersburg, FL, USA, 19–22 April 2018; pp. 1–7.
11. Di Nucci, D.; Palomba, F.; De Rosa, G.; Bavota, G.; Oliveto, R.; De Lucia, A. A developer centered bug prediction model. *IEEE Trans. Softw. Eng.* **2018**, *44*, 5–24. [CrossRef]
12. Goyal, R.; Chandra, P.; Singh, Y. Impact of Interaction in the Combined Metrics Approach for Fault Prediction. *Softw. Qual. Prof.* **2013**, *15*, 15–23.
13. Gyimesi, P.; Vancsics, B.; Stocco, A.; Mazinanian, D.; Árpád, B.; Ferenc, R.; Mesbah, A. BugsJS: A Benchmark of JavaScript Bugs. In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST), Xi'an, China, 22–23 April 2019; pp. 90–101.
14. Allen, F.E. Interprocedural Data Flow Analysis. In *Information Processing 74 (Software)*; North-Holland Publishing Co.: Amsterdam, The Netherlands, 1974; pp. 398–402.
15. Ryder, B.G. Constructing the Call Graph of a Program. *IEEE Trans. Softw. Eng.* **1979**, *SE-5*, 216–226. [CrossRef]
16. Graham, S.L.; Kessler, P.B.; Mckusick, M.K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* **1982**, *17*, 120–126. [CrossRef]
17. Xie, T.; Notkin, D. An Empirical Study of Java Dynamic Call Graph Extractors. In *University of Washington CSE Technical Report 02-12*; Department of Computer Science & Engineering University of Washington: Seattle, WA, USA, 2002; Volume 3.
18. Murphy, G.C.; Notkin, D.; Griswold, W.G.; Lan, E.S. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* **1998**, *7*, 158–191. [CrossRef]
19. Eichinger, F.; Pankratius, V.; Große, P.W.; Böhm, K. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In *Testing–Practice and Research Techniques*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 56–71.
20. Dmitriev, M. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. *SIGSOFT Softw. Eng. Notes* **2004**, *29*, 139–150. [CrossRef]
21. Eisenbarth, T.; Koschke, R.; Simon, D. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy, 7–9 November 2001; p. 602.
22. Feldthaus, A.; Schäfer, M.; Sridharan, M.; Dolby, J.; Tip, F. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In Proceedings of the 2013 International Conference on Software Engineering, Francisco, CA, USA, 18–26 May 2013; IEEE Press: Piscataway, NJ, USA, 2013; pp. 752–761.
23. Yu, L. Empirical Study of Python Call Graph. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1274–1276. [CrossRef]
24. Bolin, M. *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*; O'Reilly Media, Inc.: Newton, MA, USA, 2010.
25. Fink, S.; Dolby, J. WALA—The TJ Watson Libraries for Analysis. Available online: https://github.com/wala/WALA (accessed on 10 October 2015)
26. Antal, G.; Hegedus, P.; Tóth, Z.; Ferenc, R.; Gyimóthy, T. Static JavaScript Call Graphs: A comparative study. In Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), Madrid, Spain, 23–24 September 2018; pp. 177–186.
27. Wei, S.; Ryder, B.G. Practical blended taint analysis for JavaScript. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, Lugano, Switzerland, 15–20 July 2013; pp. 336–346.
28. Dijkstra, J. Evaluation of Static JavaScript Call Graph Algorithms. Ph.D. Thesis, Software Analysis and Transformation, CWI, Amsterdam, The Netherlands, 1 January 2014.
29. Madsen, M.; Livshits, B.; Fanning, M. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 499–509.
30. Lhoták, O. Comparing Call Graphs. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, USA, 13–14 June 2007; pp. 37–42.
31. Ali, K.; Lhoták, O. Application-Only Call Graph Construction. In *ECOOP 2012—Object-Oriented Programming*; Noble, J., Ed.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 688–712.
32. Kinable, J.; Kostakis, O. Malware Classification Based on Call Graph Clustering. *J. Comput. Virol.* **2011**, *7*, 233–245. [CrossRef]
33. Eichinger, F.; Böhm, K.; Huber, M. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *Machine Learning and Knowledge Discovery in Databases*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 333–348.
34. Rao, A.; Steiner, S.J. Debugging from a Call Graph. U.S. Patent 8,359,584, 22 January 2013.
35. Musco, V.; Monperrus, M.; Preux, P. A Large Scale Study of Call Graph-based Impact Prediction using Mutation Testing. *Softw. Qual. J.* **2016**, *25*. [CrossRef]
36. Munaiah, N.; Meneely, A. Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs. In Proceedings of the 2016 ACM Workshop on Software PROtection, Vienna, Austria, 24–28 October 2016.

37. Nguyen, V.H.; Tran, L.M.S. Predicting Vulnerable Software Components with Dependency Graphs. In Proceedings of the 6th International Workshop on Security Measurements and Metrics, Bolzano, Italy, 15 September 2010; Association for Computing Machinery: New York, NY, USA, 2010. [CrossRef]
38. Cheng, X.; Wang, H.; Hua, J.; Zhang, M.; Xu, G.; Yi, L.; Sui, Y. Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding. In Proceedings of the 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), Hong Kong, China, 10–13 November 2019; pp. 41–50.
39. Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A. Predicting Vulnerable Software Components. In Proceedings of the 14th ACM conference on Computer and Communications Security, Alexandria, VA, USA, 28–31 October 2007; pp. 529–540. [CrossRef]
40. Lee, J.; Jeong, K.; Lee, H. Detecting Metamorphic Malwares Using Code Graphs. In Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, 22–26 March 2010; Association for Computing Machinery: New York, NY, USA, 2010; pp. 1970–1977. [CrossRef]
41. Punia, S.K.; Kumar, A.; Sharma, A. Evaluation the quality of software design by call graph based metrics. *Glob. J. Comput. Sci. Technol.* **2014**, *14*, 59.
42. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 437–440. Tool demo.
43. Antal, G.; Tóth, Z.G.; Hegedűs, P.; Ferenc, R. Enhanced Bug Prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics (Training Dataset). *Zenodo*, **2020**. [CrossRef]
44. Ferenc, R.; Viszkok, T.; Aladics, T.; Jász, J.; Hegedűs, P. Deep-water framework: The Swiss army knife of humans working with machine learning models. *SoftwareX* **2020**, *12*, 100551. [CrossRef]
45. Wilcoxon, F.; Katti, S.; Wilcox, R.A. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Sel. Tables Math. Stat.* **1970**, *1*, 171–259.