[Array 15 (2022) 100189](https://doi.org/10.1016/j.array.2022.100189)

|  |  |  |
| --- | --- | --- |
|  | Contents lists available at [ScienceDirect](http://www.elsevier.com/locate/array) |  |
| Array |
| journal homepage: [www.elsevier.com/locate/array](http://www.elsevier.com/locate/array) |
|  | | |

Using textual bug reports to predict the fault category of software bugs Thomas Hirsch, Birgit Hofer∗  
*Institute of Software Technology, Graz University of Technology, Austria*

|  |  |  |
| --- | --- | --- |
| A R T I C L E | I N F O | A B S T R A C T |
| *Keywords:*  Bug report  Bug benchmark  Fault type prediction |  | Debugging is a time-consuming and expensive process. Developers have to select appropriate tools, methods and approaches in order to efficiently reproduce, localize and fix bugs. These choices are based on the developers’ assessment of the type of fault for a given bug report. This paper proposes a machine learning (ML) based approach that predicts the fault type for a given textual bug report. We built a dataset from 70+ projects for training and evaluation of our approach. Further, we performed a user study to establish a baseline for non-expert human performance on this task. Our models, incorporating our custom preprocessing approaches, reach up to 0.69% macro average F1 score on this bug classification problem. We demonstrate inter-project transferability of our approach. Further, we identify and discuss issues and limitations of ML classification approaches applied on textual bug reports. Our models can support researchers in data collection efforts, as for example bug benchmark creation. In future, such models could aid inexperienced developers in debugging tool selection, helping save time and resources. |

**1. Introduction**

Software bugs consume a significant amount of software developer resources, resulting in financial losses [1]. For these reasons, sizeable research fields developed around bugs in software, providing novel and advanced tools and approaches. In real world software develop-ment, this progress is most prevalent in the application of ante-mortem approaches that aim at preventing the introduction of bugs, as for example bug prediction, static checking, and automated testing. However, once a bug has manifested itself and been reported, developers mostly utilize classic debugging tools and techniques to reproduce, localize and fix bugs [2–4]. A wide variety of such debug-ging tools and techniques are available to developers, e.g., breakpoints, conditional breakpoints, memory profilers, and leak detection tools. While some of these tools are highly specialized towards specific fault types and steps of the debugging process, other approaches cover a broader spectrum of possible applications in the debugging process. However, there is no one fits all solution.

The developers’ choices of debugging approaches and tools there-fore have a great influence on their success and pace in fixing the bug. In most cases, this choice depends on the information obtained from the textual bug reports, and the developers’ experience and knowledge. In order to encapsulate and discretize some of this knowledge, we have created a bug classification schema on a high abstraction level. We categorize bugs according to their underlying fault into *Concur-rency*, *Memory*, *Semantic*, and *Other* bugs. With these categories, we

try to provide an abstraction that encapsulates the different challenges developers face when dealing with a specific bug type including the different tools and approaches for reproduction and localization. To leverage on such a classification schema for practical debugging sup-port, e.g., for tool recommendation, a priori knowledge about a bug’s type is required.

In this paper, we propose a machine learning (ML) based classifier that predicts the fault type of a bug based on its textual bug report. This ML based approach tries to encapsulate a small aspect of what would commonly be considered developer knowledge and experience in debugging. A priori information about the underlying fault type can support inexperienced developers in their choice of debugging approaches and tools.

This article extends previous work [5] presented at the *4th Interna-tional Workshop on Software Faults* in the following ways: (1) We extend the employed classification schema by the *Other* category. (2) We enlarge the training set by 127 bug tickets and we quantify the quality of the training set with internal and external verification methods. (3) We perform a user survey to establish a baseline of human clas-sifier performance on this task. (4) We introduce preprocessing steps tailored specifically for bug reports and we apply ensemble learning methodologies on the classification problem. (5) We evaluate the classi-fication performance of various classical ML algorithms, preprocessing steps, and ensemble approaches. (6) We evaluate the performance for inter-project application. Our main findings of this Journal version are:

∗ Corresponding au[thor.](mailto:thirsch@ist.tugraz.at)

*E-mail addresses:* [thirsch@ist.tugraz.at](mailto:thirsch@ist.tugraz.at) (T. Hirsch), [bhofer@ist.tugraz.at](mailto:bhofer@ist.tugraz.at) (B. Hofer).

<https://doi.org/10.1016/j.array.2022.100189>  
[Received 8 March 2022; Received in revised](https://doi.org/10.1016/j.array.2022.100189) form 11 May 2022; Accepted 13 May 2022 Available online 26 May 2022

2590-0056/© 2022 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

• Non-expert human classifier performance averages at 0.62 weighted average F1.

• Different ML algorithms show significantly different performance on specific classes (e.g., *Memory* bugs on Multinomial Naive Bayes and Random Forrest classifiers, with 0.72 and 0.79 macro average F1 respectively.)  
• Ensemble approaches perform (0.69 macro average F1) better than the best performing single classifier model (0.66 macro average F1 for Logistic Regression classifier), with ensemble size playing only a minor role.

The remainder of this paper is structured as follows: Section 2 describes the problem of fault type prediction using bug reports. In Section 3, we discuss related work and similar classification endeavors. In Section 4, we discuss the background of this work including exist-ing bug classification schemata, ML based classification, and natural language processing (NLP). In Section 5, we present our experimental setup and approach, followed by our research questions and results in Section 6. In Section 7, we discuss the internal and external threats to validity. Section 8 concludes our work and discusses future research. All datasets and implementations are publicly available (see Section 8).

**2. Problem**

The classic NLP example of sentiment analysis is based on the assumption that ‘sentiment’ information is inherent to human written text, and the assumption that the inputs contain only human written text. Bug reports are very different from such showcase NLP prob-lems. The information required for correct classification may not be contained in a bug report: Bug reports are textual descriptions of complicated behaviors, states, and outcomes to communicate a problem in a complex system. Such reports are authored by people occupying different roles, functions, and positions in relation to the project. These roles range from end-users unfamiliar with software development, to highly experienced developers with years of experience within the project. Because of this, the authors’ technical expertise and project specific expertise can vary significantly. Further, different roles have inherently different viewpoints and scopes of the bug reports. For example, bug reports may describe only the bugs’ impact in non-technical terms, while other bug reports may describe a problem in technical detail without mentioning possible impacts.

In addition, bug reports can take many shapes due to a wide variety of templates and formatting rules, or lack thereof, as well as the noncompliance to such rules. This results in noisy,1and sometimes incomplete bug reports, that widely vary in length,2content, and vocabulary.

In some cases, deriving aforementioned fault type from such bug reports can be trivial, as for this *memory* bug3:

|  |  |
| --- | --- |
| |  | | --- | | **Native (java) process memory leak** An internal memory leak when using GarbageCollectorMXBean#getLastGcInfo in the JVM. Disable using it... | |

However, more often than not, it can be challenging the readers’domain and project specific knowledge and expertise, as for example this *semantic* bug originating from *Spring Boot*4:

|  |  |
| --- | --- |
| |  | | --- | | **Gradle plugin still includes \*Launcher classes with Layout.NONE** *No description provided.* | |

In some cases, it is even impossible, e.g., this bug report, arising from a documentation error5:

1 see e.g., [orientdbissue2121.](https://github.com/elastic/elasticsearch/issues/4960)

2 [see e.g.,](https://github.com/elastic/elasticsearch/issues/1118) [elasticsearchissue4960](https://github.com/elastic/elasticsearch/issues/4960).   
3 [elasticsea](https://github.com/spring-projects/spring-boot/issues/139)[rchissue1118.](https://github.com/elastic/elasticsearch/issues/4960)

4 [spring-bootissue139](https://github.com/spring-projects/spring-boot/issues/139)[.](https://github.com/elastic/elasticsearch/issues/1118)

5 [nettyissue1508.](https://github.com/spring-projects/spring-boot/issues/139)

2

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

a given bug report and to extract patterns from large code bases [20]. A recent survey [21] provides a good overview on the application of deep learning approaches in software engineering research, including defect and vulnerability prediction and bug localization.

Particularly interesting in the context of fault localization is the approach proposed by Fang et al. [22] that classifies bug reports as informative or uninformative. This approach can be used as a prepro-cessing step in information retrieval (IR) approaches to filter out those bug reports where IR promises little insights.

Huang et al. [23] manually labeled the intention of more than 5,400 sentences from issue reports into seven categories, e.g. ‘Information Giving’ and ‘Problem Discovery’. Afterwards, they trained a deep neural network to predict these intentions.

**4. Background**

First, we provide an overview of existing bug classification schemata (Section 4.1). Afterwards, we briefly explain the used classifiers, and statistical methods (Section 4.2) and the most important terms w.r.t. natural language processing (Section 4.3). Finally, we provide the formal definitions of the used performance metrics (Section 4.4).

*4.1. Bug classification schemata*

A plethora of bug, fault, and defect classification schemata has been proposed in the past by researchers and practitioners alike. Such a classification schema can cover one or more dimensions of a defect, e.g. severity, impact, and root cause. All classification schemata are of course intended to fulfill a certain purpose, and their dimensions, depth, and detail are selected to achieve the set goal. These purposes range from investigations into process optimization to enable auto-mated triage and prioritization, to research into different areas of the software development process, to the support of techniques such as automated repair.

Polski et al. [24] discussed the application of existing fault classifi-cation schemata and bug distributions for fault injection, and provided an overview on fault classification schemata. Endres [25] performed one of the earliest attempts at bug classification to investigate higher level causes (e.g., technological, organizational, historic).

Gray [26] introduced the categories *Bohrbugs* and *Heisenbugs*. Grot-tke and Trivedi [27] extended upon Gray’s categories by introducing *Mandelbugs* and *aging-related bugs*.

Chillarege et al. [28] devised the Orthogonal Defect Classification (ODC) schema to form the basis for analysis and optimization of a soft-ware development process. The *IEEE Standard Classification for Software Anomalies (IEEE Std 1044–2009)* [29] has established a vocabulary for software anomalies as well as a classification schema and attributes for defects and failures.

Only a few classification schemata target the debugging process with the purpose of supporting software developers. Li et al. [9] and Tan et al. [8] studied the characteristics of bugs in open source software to enable more effective debugging tool design and better under-standing of bugs occurring in the real world. Given this focus on debugging tools and debugging processes, they classified bugs along three axes: *impact*, *software component*, and *root cause*. Impact consists of six categories (e.g., incorrect function, crash, or hang). Their root cause dimension comprises three categories: *Memory* bugs arise from improper memory handling, *concurrency* bugs occur in multi-threaded programs due to synchronization issues, including race conditions and deadlocks, and *semantic* bugs are inconsistencies between requirements or programmers’ intentions and the actual software function.

*4.2. Machine learning*

In this work, supervised machine learning approaches are applied to perform automated classification. The problem at hand is a multi-class classification problem, where each input data instance belongs to a specific class.

3

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

occurrences of the term in this document, times the inverse document frequency *𝑖𝑑𝑓* of the term.

|  |  |
| --- | --- |
| *𝑇 𝑓* − *𝑖𝑑𝑓*(*𝑡, 𝑑*) = *𝑡𝑓*(*𝑡, 𝑑*) ∗ *𝑖𝑑𝑓*(*𝑡*) | (1) |
| *𝑖𝑑𝑓*(*𝑡*) = *𝑙𝑜𝑔* ( *𝑑𝑓*(*𝑡*) + 1 *𝑛* + 1 ) + 1 (2)  The document frequency *𝑑𝑓*(*𝑡*) is the number of documents in the cor-pus that contain the term *𝑡*. The resulting Tf-idf weighted vectors are of Euclidean norm. The formula for *𝑖𝑑𝑓* above depicts the implementation in the ML software library used in this work, which differs slightly from its textbook definition [30] (see [31] for a detailed explanation). **Stemming** replaces words with their word stem. For example, the words ‘crash’ and ‘crashing’ have the stem ‘crash’. The resulting stem does not necessarily have to be a word itself. Without prior stemming,‘crash’ and ‘crashing’ are considered different tokens and therefore result in separate features in a bag of words approach; with stemming both words are represented in the same feature.  **Case folding** is performed to provide a document with all letters in the same case. Vectorizers are based on string comparison, occurrences of the same word starting with an upper case letter would be repre-sented as a different feature than the same word starting with a lower case letter.  **Decamelcasing** splits camelcased words into several words that allows for tokenization and stemming of those words.  **Stop word removal** erases very common words (e.g. ‘the’, ‘and’,‘is’) that do not add value for the task at hand from the input text. **Artifact removal** discards non-human language artifacts such as stack traces, code snippets, config files, file listings, log outputs and thread dumps from bug reports. Some bug reports in our dataset are as big as 80 kb of text because of such artifacts. Amongst the biggest artifacts are stack traces. While those traces support developers in their debugging efforts, the contained information for classification purposes is mostly limited to name and type of the occurred exceptions.  The wide variety of different formats of artifacts poses a significant problem for automated removal using regular expressions [32]. We therefore employ our custom ML based artifact removal process [33]. The underlying ML model is trained on software projects’ documenta-tion files as well as issue tickets and leverages GitHub markdown for automated training set creation. The resulting classifier model operates on a line-by-line basis and keeps exception names that occur in the artifact. | |

*4.4. Performance metrics*

We use Precision (P), Recall (R), F1-score (F1) in single class ex-aminations to measure and rank our classifiers’ performance, and to enable inter-classifier comparison, and their weighted average F1-score (waF1) and macro average F1-score (maF1) to compare multi-class performance. These metrics can be calculated from the classifiers’confusion matrices. True Positives (TP) is the number of instances in the predicted class that match the actual class. False Positives (FP) ex-presses the number of instances in predicted class that do not match the actual class. True Negatives (TN) is the number of instances correctly identified as not belonging to the class, and False Negatives (FN) is the number of instances incorrectly identified as not belonging to the class. **Precision** for a class is TP divided by the total number of instances classified to belong to this class (*𝑃* = *𝑇 𝑃* +*𝐹𝑃*). A precision of 1.0 means that all instances labeled as class X are correct;

**Recall** for a single class is TP divided by the total number of instances actually belonging to this class (*𝑅* = *𝑇 𝑃*+*𝐹𝑁*). A recall of 1.0 means that all instances of actual class X are correctly labeled as belonging to class X.

**F1** for a single class is the harmonic mean of the class’ precision and recall (*𝐹1* = 2 ∗*𝑃*∗*𝑅 𝑃* +*𝑅*).

4

|  |  |  |  |
| --- | --- | --- | --- |
| *T. Hirsch and B. Hofer* | **Table 1**  Fault type classification schema with top-level categories in bold font and 2nd-level categories in normal font. | | *Array 15 (2022) 100189* |
|  | Fault type | Description |  |
|  | **Concurrency**  Order violation  Race condition  Atomic violation  Deadlock  Parallelization  Other | **Improper or incorrect synchronization and wrong assumptions about atomicity.**  Incorrect order caused by missing or incorrect synchronization or thread handling.  Two or more threads simultaneously access the same resource with at least one being a write access.  Atomicity of a operation was wrongly assumed, or improper use of not thread safe data types and structures.  Two or more threads stuck in waiting for the other to release a resource. Missing parallelization resulting in lagging, or removal of parallelization to meet constraints.  Concurrency-related faults that do not fall in any of the above categories. |  |
|  | **Memory and resources** Overflow  Null pointer dereference Uninitialized memory read Leak  Dangling pointer  Double free  Other | **Improper or incorrect memory and resource handling.**  Buffer overflows, excluding overflowing numeric types.  Null pointer dereferences.  Uninitialized memory reads except null pointer dereference.  Memory and resource leaks.  Pointers referring to invalid data.  Freeing the same memory more than once.  Memory-related faults that do not fall in any of the above categories. |  |
|  | **Other**  Documentation  Build system  UI resources  Configuration | **Bugs that cannot be resolved by changing Java code.**  Incorrect or missing documentation.  Missing or incorrect build configuration and bugs in build scripts. Missing or incorrect UI resources.  Missing or incorrect config files, excluding build config and UI config. |  |
|  | **Semantic**  Exception handling Missing case  Processing  Typo  Dependency  Other | **Inconsistency of requirements, programmers intentions, and actual implementation that do not fall in the above categories.**  Incorrect, missing, or overshooting exception handling.  Code is missing. Missing implementation, missing program flow, or missing other code parts due to unawareness of a certain case.  Code is incorrect. Incorrect implementations, ranging from miscalculations to incorrect library usage.  Simple typographic errors.  Bugs introduced by changes in a foreign library or system that lead software that can be built, but behaves unexpected or incorrect.  Faults that do not fall into any of the above categories. |  |

• changed more than 250 LOC or more than 20 files per commit,• link to at least one commit that has become unavailable,  
• have commits with more than one parent, or  
• have commit messages mentioning more than one issue ticket id or containing phrases that indicate multiple fixes (e.g., ‘also fixes’).

This reduced dataset consists of 11 621 bug reports and forms the basis for our experiments.

*5.3. Trainingset creation*

*Managing imbalance.* This raw collection of issues tickets is ex-pected to be highly imbalanced w.r.t. our classification target. Other re-searchers identified *Memory* and *Concurrency* bugs as minority classes, making up only 2%-16% of all bugs [8–10]. Since our selected ML algorithms are sensitive to such imbalance, we will employ down-sampling to balance our dataset. However, a certain number of data points for the minority classes are required, as the size of these minority classes dictate the resulting training set size. We estimate that our dataset of 11 621 bug tickets contains only a few hundred *Memory* and *Concurrency* bugs. Since exhaustive examination of all issues is deemed infeasible, we need to filter and preselect issues for manual examination.

We therefore performed a keyword search on commit messages to identify candidates for manual classification. We used a modified version of the keywords used by Ray et al. [10]. Our keyword set contains 29 keywords and regular expressions, e.g., ‘overflow’, ‘\sleak’,‘deadlock’, ‘\shangs\s’, ‘\sstarves\s’. The complete list of keywords is available in the online appendix (see Section 8). Commit messages are authored by the developers performing the bug fix. These developers

5

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

*Dataset quality and verification.* Researcher 1 performed the manual classification described above. Six months later, Researcher 1 reclassi-fied 100 randomly sampled and blinded issues from the set for internal verification. In this internal verification step, Researcher 1 scored 0.95 for Cohens Kappa, that suggests *almost perfect agreement* [35] and a weighted average F1 of 0.96.

For external verification, Researcher 2 classified 246 randomly sampled and blinded issues. This external verification resulted in a Cohens Kappa score of 0.69, suggesting *substantial agreement* [35], and a weighted average F1 of 0.80.

*Final data sets.* In preliminary ML experiments, we identified three projects that are not suitable for our approach of fault type classifi-cation. These projects are: *LeakCanary*, a Java memory leak detection library, *Bazel*, a build automation framework, and *JHipster*, a web application generator. Their domains make it impossible to correctly identify certain bug types using an NLP approach based only on bug reports without knowledge on the projects’ domain and purpose. For example, bug reports from a memory leak detection library utilize a vocabulary otherwise directly connected to memory leaks for all classes of bugs, further reinforced by class names and function names within these software projects. Analog to this, bug reports from a build framework have a vocabulary otherwise associated with *Other* bugs in any general purpose software project. We therefore removed all items from these projects from the training/test sets and survey answers. We used the initial manual classifications of Researcher 1 as the basis for the experiments and balanced the resulting data set by un-dersampling the majority classes, resulting in equally sized categories containing each 124 bug reports. The final data set used in our exper-iments has a total size of 496 bug reports from 71 different software projects.

*5.4. ML classifier*

*ML algorithms.* We have selected Logistic Regression (LR), Multino-mial Naive Bayes (MNB), Random Forrest (RF), and Support Vector Machines (SVM) based on other researchers’ work in similar endeav-ors [6,8,9,17,36], and their ease of use. Further, we employ a LR-based stacking classifier ensemble learning approach to combine the best performing models.

*Experiment setup.* Our models’ pipelines consist of artifact removal, and decamelcasing, followed by stemming and count vectorization, through Tf-idf, to finally a classifier algorithm. We perform nested CV with five folds each for hyperparameter tuning and model selection. The inner *𝑘*-fold CV is used for hyperparameter tuning of the models. The selected hyperparameter space only concerns preprocessing steps, and consists of enabling or disabling stop word removal, decamelcas-ing, stemming, artifact removal, and usage of Tf-idf. Parameters of the classifier algorithms are not tuned and kept at their default values (scikit-learn 0.24.2). The outer *𝑛*-fold is used for model selection. All k-fold CV splits used in our experiments were done in a stratified manner to conserve the balanced nature of the input dataset.

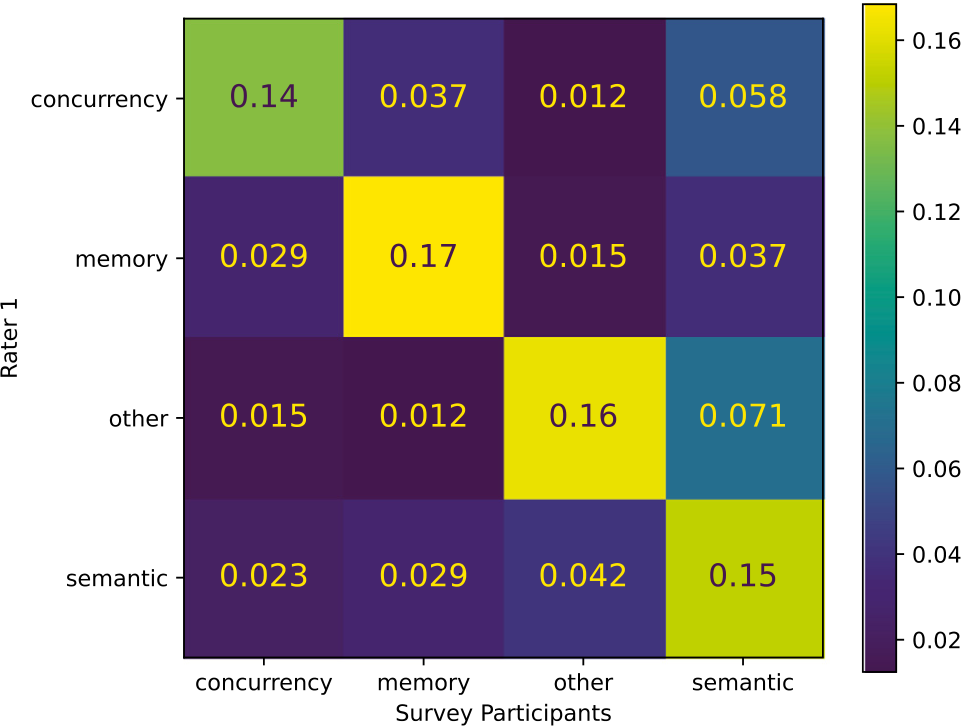
To evaluate the resulting models, we use Bootstrap with stratified 80–20 training-test splits, *𝑛* = 100 repetitions and a confidence level of *𝛼*  = 0*.*95. We calculate the mean performance scores from all Bootstrap repetitions. For comparing two models, we perform one-sided T-tests on the scores from both models’ Bootstrap repetitions with *𝐻*0 = *𝑀𝑜𝑑𝑒𝑙 𝐴 𝑖𝑠 𝑛𝑜𝑡 𝑏𝑒𝑡𝑡𝑒𝑟 𝑡ℎ𝑎𝑛 𝑀𝑜𝑑𝑒𝑙 𝐵*. Further, we build confusion matrices through aggregation of the Bootstrap iterations’ confusion matrices.

**6. Experimental results**

*6.1. RQ1: What is the performance of humans classifying bug reports?*

**Motivation:** By answering RQ1, we establish a baseline to compare ML approaches against.

6



*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

**Fig. 2.** Confusion matrix of all received answers.

report level, we can only speculate over the reasons for this rather low overall performance. However, issues such as incomplete, ambiguous, or even misleading bug reports are suspected to play a significant role in this. Netty issue 15086for example, was classified as *memory* bug by our participants, while the subsequent discussion and fixing commit on GitHub clearly show it to be a documentation error and therefore belonging into the *other* category.

**Answer:** Based on these findings, we establish a human (non-expert) performance baseline of 0.62 weighted average F1.

*6.2. RQ2: What performance is achievable using ML algorithms for classi-fying bug reports?*

**Motivation:** We investigate the performance of classic ML ap-proaches for fault type classification. Despite this narrow focus, cap-turing only a small but essential aspect of a bug, the obtained results can serve as an indicator of the applicability of classic ML approaches on textual bug reports for technical debugging support.

**Approach:** We perform three consecutive experiments, measure the performance of classifier models, and identify benefits and drawbacks of preprocessing steps and classifier models. Further, we investigate misclassifications and their reasons.

**EXP1:** We performed nested CV with an SVM classifier algorithm to establish a performance baseline. The best scoring model of the outer CV was then evaluated using Bootstrap. We performed this experiment twice, with and without utilizing artifact removal.

**Results:** Fig. 3(a) shows the macro average F1 Bootstrap confidence intervals and mean performance of the two models. Artifact removal increases the classifier’s mean macro average F1 from 0.62 to 0.65. A one sided T-test of the models’ performance scores confirms that the model with artifact removal is better than the model without this preprocessing step (*𝑝* = 4 ∗ 10−10).

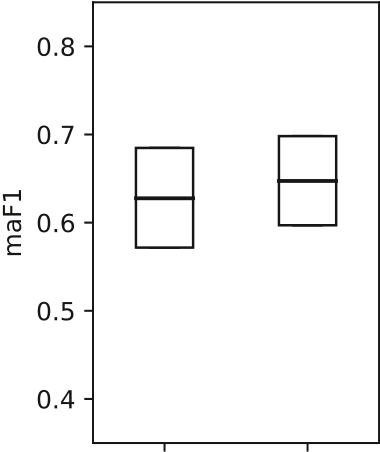
However, this increase in overall performance is not reflected in in-creased performance for each bug class as can be observed in Fig. 3(b). While for example, the model with artifact removal performs signifi-cantly better than the model without artifact removal for *Memory* bugs (*𝑝* = 2 ∗ 10−30) with mean F1 scores of 0.74 vs. 0.66, it is the other way round for *Semantic* bugs (*𝑝* = 0*.*001) with mean F1 scores of 0.48 and 0.66.

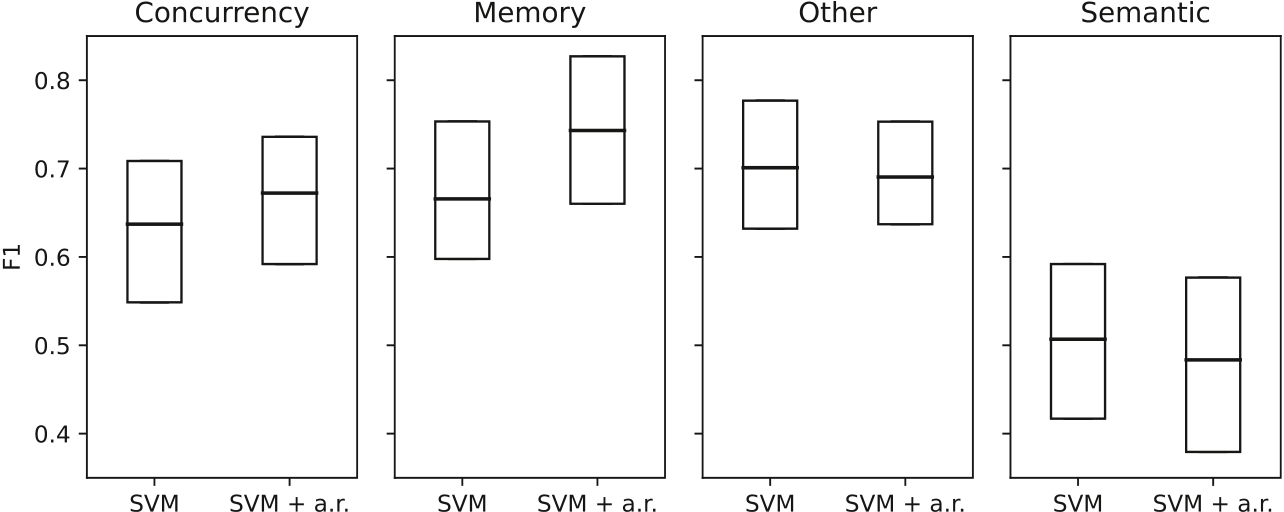
**EXP2:** We performed nested CV with MNB, SVM, RF, and LR classifier algorithms. We again selected the best scoring model from the outer CV and evaluated them using Bootstrap.

6 [nettyissue1508](https://github.com/netty/netty/issues/1508).

7

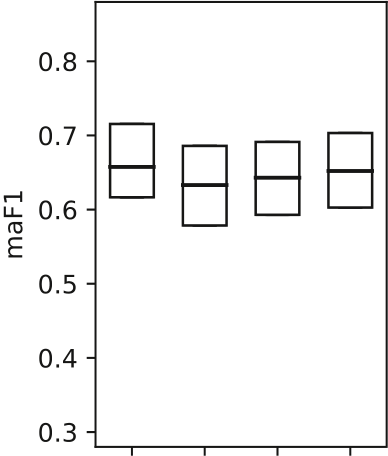
*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

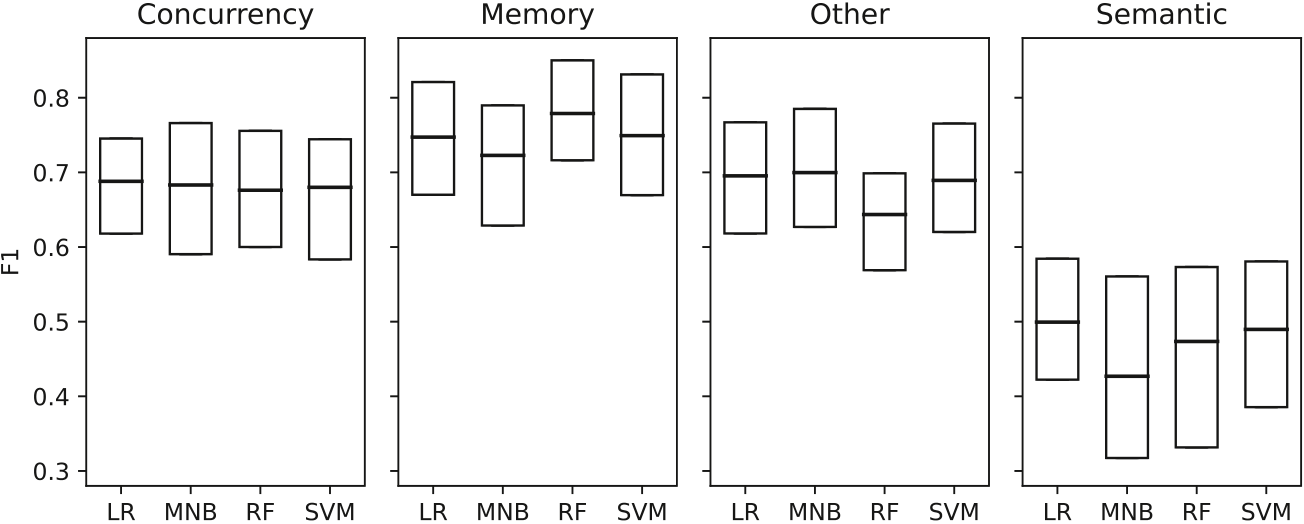


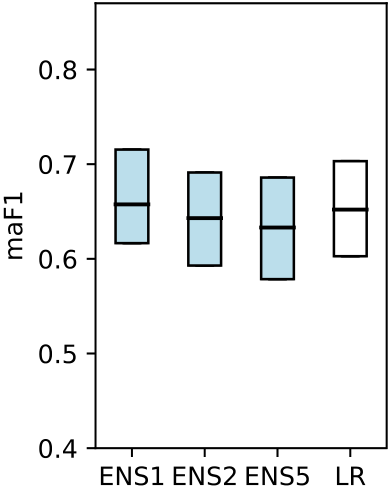
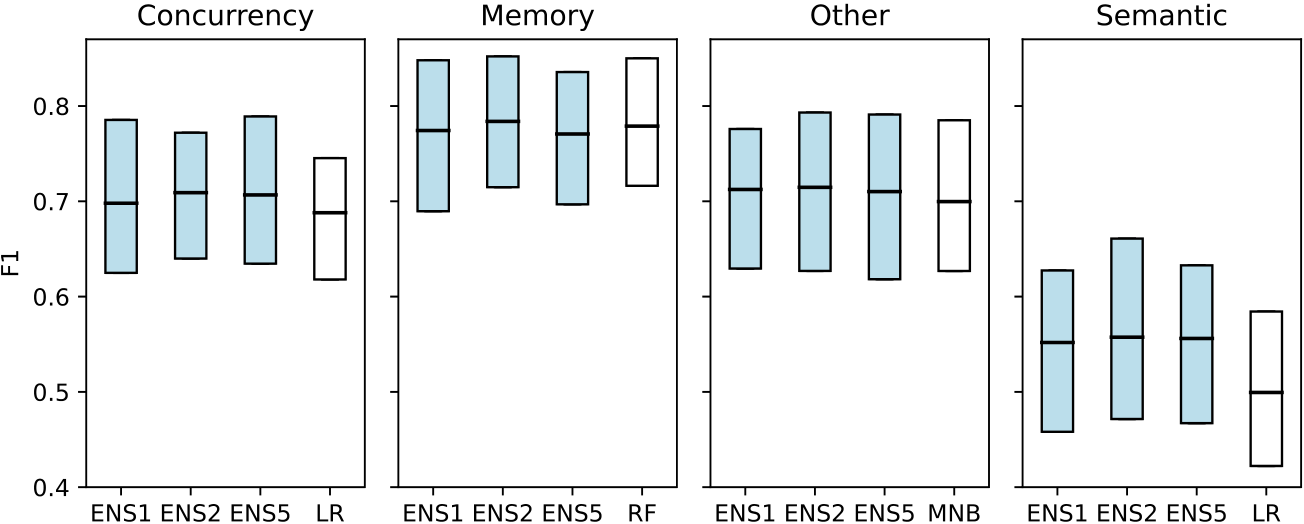
**Fig. 3.** EXP1: Bootstrap confidence intervals for SVM and SVM with artifact removal.



**Fig. 4.** EXP2: Bootstrap confidence intervals for best performing model from each classifier algorithm.

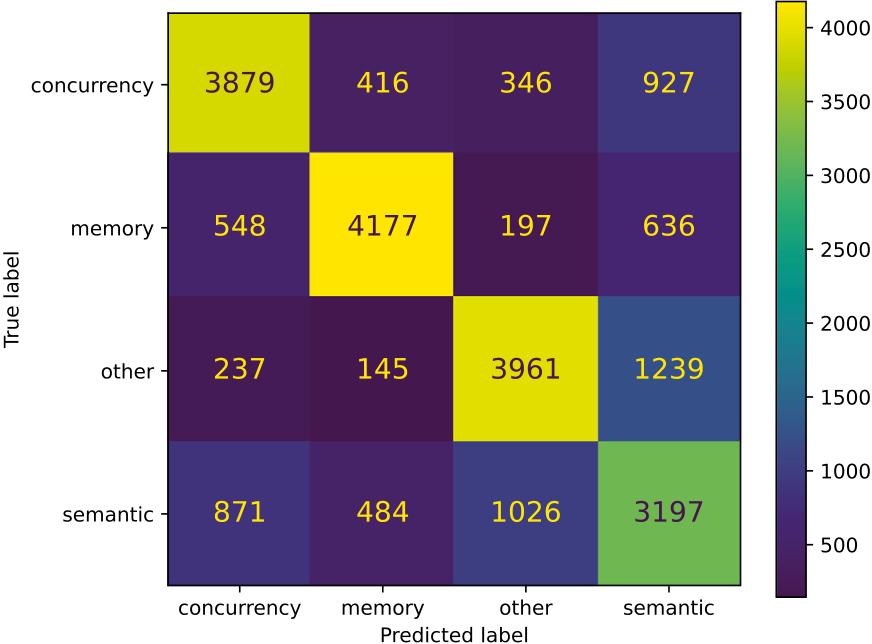
**Fig. 5.** EXP3: Bootstrap confidence intervals of the ensembles and of best performing classifier models from Experiment 2.

were incorrect predictions, and the remaining 15214 predictions were correct.

We compared the length in characters and length in lines, for both the original bug reports, and the bug reports after artifact removal, as well as the lengths in characters and lines of removed artifacts. Further, we compared the number of occurrences of exception names in the bug tickets, as well as the number of bug tickets that contain such exception names. However, none of these metrics show any significant differences between the misclassifications and correct classifications, which can be explained by significant overlap of contained bug tickets in both correct and incorrect sets because of the small original dataset size of 496 bug tickets.

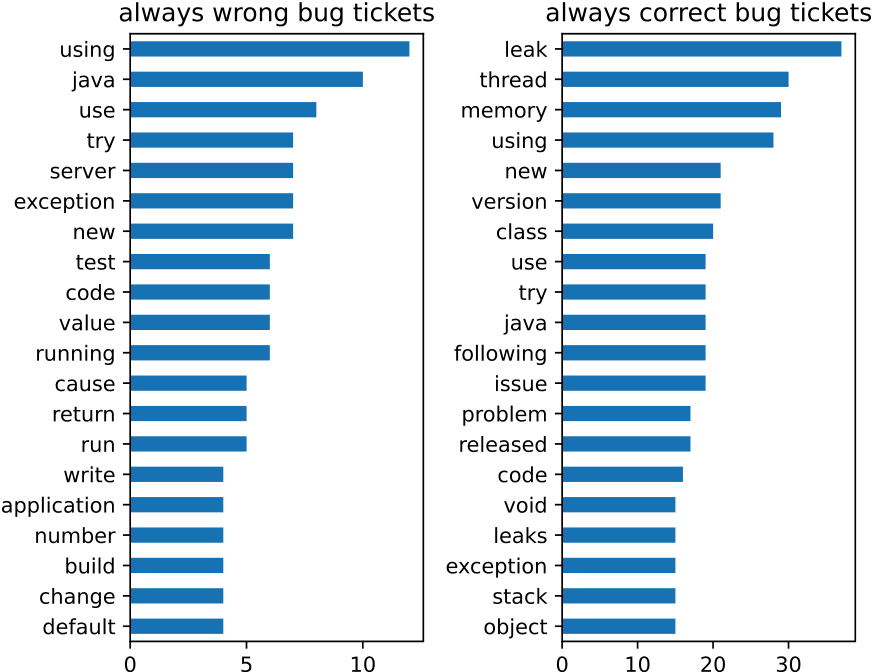
We therefore investigated bug tickets that were always predicted correctly (141), or always predicted incorrectly (33). While above listed metrics hint towards shorter tickets with shorter non-human written

8



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *T. Hirsch and B. Hofer* | | | | |  |  |  |  |  | *Array 15 (2022) 100189* |
| **Table 2**  Hyperparameters used in ensemble classifiers. | | | | |
| Classifier | | | | Target | Artifact  removal | Class  weight | Decamel-case | Tf-idf | Stemm-ing | Stop word  removal |
|  | mble 2 | Ensem. 1 | SVM  RF  MNB  RF | Concurrency Memory  Other  Semantic | true  true  false  false | 0\_4  1\_4 – 3\_4 | false  true  false  true | true  true  true  true | true  false  false  true | false  English  English  false |
| 3 | Ense |  | RF  MNB  MNB  MNB | Concurrency Memory  Other  Semantic | false  true  true  false | 0\_4 – – – | true  false  false  false | true  false  true  false | true  true  true  true | English  English  English  false |
| Ensemble 3 |  |  | MNB  MNB  MNB  SVM  RF  RF  MNB  SVM  SVM  SVM  RF  RF | Concurrency Concurrency Concurrency Memory  Memory  Memory  Other  Other  Other  Semantic  Semantic  Semantic | true  true  true  true  true  true  true  false  false  false  false  false | – – – 1\_4  1\_4  1\_4 – 2\_4  2\_4  3\_4  3\_4  3\_4 | false  false  false  false  true  true  true  true  true  true  true  true | false  false  false  true  true  true  true  true  true  true  true  true | true  true  true  true  false  true  true  true  true  true  true  true | English  English  English  false  English  English  false  false  false  false  false  English |

**Fig. 6.** EXP3: Cumulative confusion matrix from 100 Bootstrap iterations for Ensemble 1.



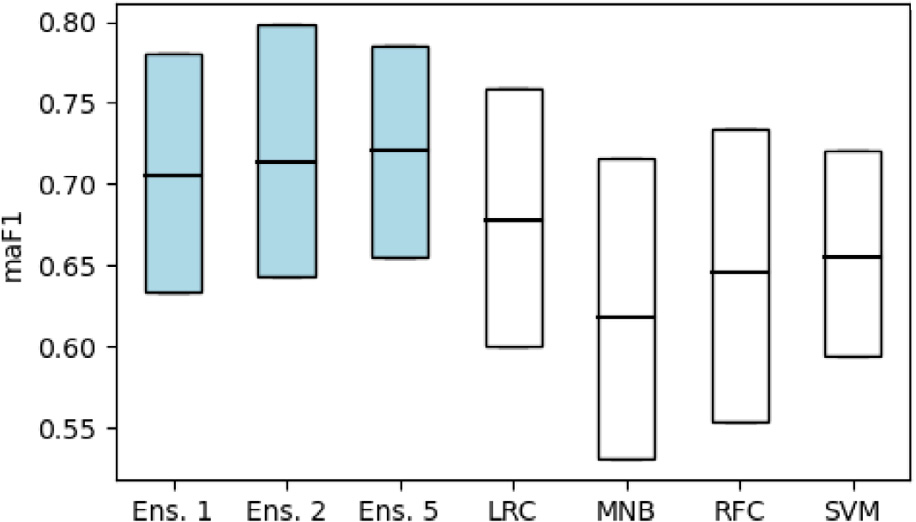
 

**Fig. 7.** EXP3: Words contained in always incorrectly and always correctly predicted bug reports with Ensemble 1.

**Discussion:** Our results show that preprocessing steps as artifact removal, although increasing overall performance, have a detrimental

9

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*



**Fig. 8.** 95% percentiles of F1 macro average scores for 100 project splits.

Analog to our approach in RQ2, we perform nested CV for each classifier algorithm (SVM, LR, RF, MNB) and for each category while adding weights to the respective category using only the training set. We take the best *𝑚* models from each category and combine the selected models by ensembling them using a stacking classifier ensemble with a LR base classifier. Finally, we train the resulting ensemble on the training set and evaluate its performance on the validation sets. **Results:** Due to the great class imbalance found within most projects, some projects appear more often than others in these project splits, while some projects do not appear at all. *Netty* with a total of 84 bug tickets does not occur in any validation split due to its high number of memory bugs that collides with our balancing and size criteria discussed above, and *elasticSearch* occurs in 78 project splits due to its big size (41 bug reports) and being almost balanced itself. Fig. 8 shows the 95% percentiles and mean of macro average F1 scores as boxplots. Again, the mean macro average F1 performances of the ensembles are closely clustered, ranging from 0.71 for Ensemble 1 to 0.72 for Ensemble 3. One sided T-test on the ensembles’ scores shows that we cannot claim an increase of performance due to ensemble size (*𝑝* = 0*.*06 for Ensemble 2 scores to be not greater than Ensemble 1 scores, and *𝑝* = 0*.*09 for Ensemble 5 scores to be not greater than Ensemble 2 scores). However, Ensemble 1 with a mean macro F1 score of 0.71 performs significantly better than the single classifier models LR (0.68) and SVM (0.65). Results for one sided T-test for Ensemble 1 scores being greater than LR and SVM classifiers are *𝑝* = 1 ∗ 10−6and *𝑝* = 2 ∗ 10−18.

We calculate the project specific F1 score from the collective test sets and their predictions from all iterations where this project oc-curred. Since the issues from separate projects are imbalanced w.r.t the amount of bugs for each fault type, we use the weighted average F1 score as metric. Again, we only consider projects with at least ten issue tickets contained in the training set. Table 3 shows the project specific scores for these projects, the entry *ALL OTHERS* contains all the remaining projects with less than 10 bug reports. The highest weighted average F1 score of 0.93 can be found for *spring-framework* (11 bug reports); the lowest score of 0.55 occurs for *n4js* (12 bug reports). We analyzed bug reports meta data, as document lengths, distribution of contained fault types, occurrence of artifacts of the projects listed in Table 3, however, we did not find any correlations to the projects’performance scores. Further, the projects’ bug tickets sample sizes are too small to draw any conclusions. However, we manually investigated the bug reports of the best (*spring-framework*) and the worst (*n4js*) performing projects. We found that *spring-framework* bug reports are easier to classify by humans as they contain the information required in contrast to more ambiguous bug reports in *n4js*.7

|  |  |
| --- | --- |
| 7 <https://github.com/spring-projects/spring-framework/issues/24265><https://github.com/eclipse/n4js/issues/444>[.](https://github.com/spring-projects/spring-framework/issues/24265) | vs. |

10

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *T. Hirsch and B. Hofer* | **Table 3**  Ensemble 1 performance on specific projects bugs in cross project evaluation. | | | |  |  |  | *Array 15 (2022) 100189* |
|  | Project | Bug  reports | Bug reports occurred | Splits contain-ing project | Precision (w.a.) | Recall  (w.a.) | F1  (w.a.) |  |
|  | spring-framework spring-security  redisson  junit5  ExoPlayer  spring-session  hazelcast  async-http-client ALL\_OTHERS  elasticsearch  checkstyle  spring-boot  n4js | 11  20  25  12  12  15  16  14  185  41  14  35  12 | 165  180  1325  456  324  680  448  448  1349  3198  196  455  264 | 15  9  53  38  27  40  28  32  100  78  14  13  22 | 0.94  0.86  0.88  0.93  0.84  0.79  0.76  0.79  0.68  0.67  0.72  0.72  0.60 | 0.93  0.85  0.83  0.82  0.83  0.78  0.71  0.69  0.66  0.65  0.67  0.58  0.59 | 0.93  0.85  0.85  0.85  0.84  0.78  0.72  0.71  0.67  0.65  0.63  0.58  0.55 |  |

to other programming languages, issue trackers, and closed source software.

We utilized labels on GitHub issue trackers to select candidates for our dataset. Labeling is performed manually by the software project maintainers, and is therefore subject to misclassifications. We counter this threat by excluding bug reports that we deem mislabeled, e.g., fea-ture requests wrongly labeled as bugs.

The manual classification performed by researcher 1 is also subject to misclassifications and therefore a threat to internal validity. To counter this threat, a second researcher independently classified a blinded random sample of the dataset, and researcher 1 re-classified a blinded random sample six months after the initial classification. We used these additional samples to calculate inter-rater agreement scores, to quantify the quality of our dataset.

The majority of participants for our survey are master degree stu-dents in computer science. These participants are to be considered non-experts, as they are not involved in the development of the soft-ware projects sourcing our datasets. Further, participants performed classification without provision of the original project context. The resulting scores are therefore on the lower end of human classification performance for the given task.

**8. Conclusion**

We have investigated the application of NLP and classical ML ap-proaches on textual bug reports to predict the bug type in terms of four classes, *Concurrency*, *Memory*, *Other*, and *Semantic* bugs.

We have investigated human classifier performance on this task, followed by experiments using classical ML algorithms on this multi-class classification problem. The mean classification performance of non-expert human classifiers was rather low, with a mean weighted average F1 score of 0.62. The best single classifier model (Logistic Regression) has 0.66 macro average F1.

Our investigation into ML approaches highlighted advantages and disadvantages of certain NLP preprocessing steps and different ML algorithms. To exploit the gained insights, we used ensemble methods that combine multiple classifier models and preprocessing pipelines. Using such ensemble methods, we achieved mean macro average F1 scores of 0.69.

Not all types of bugs are equally hard to predict, and our models parallel the strengths and weaknesses of human classifiers: *Memory* bugs are the class with the highest classification performance for hu-mans (0.63 mean weighted average F1 score), as well as standalone classifier models (0.79 mean macro average F1 for Random Forrest clas-sifiers), and ensemble models (0.77 mean macro average F1). *Semantic* bugs constitute the class with the lowest classification performance for humans (0.47 mean weighted average F1 score), standalone classifier models (0.55 mean macro average F1 for Multinomial Naive Bayes classifiers), and our ensemble models (0.55 mean macro average F1).

11

*T. Hirsch and B. Hofer*  *Array 15 (2022) 100189*

**Declaration of competing interest**

The authors declare that they have no known competing finan-cial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Acknowledgment**

The work described in this paper has been funded by the Austrian Science Fund (FWF): P 32653 (Automated Debugging in Use).

**References**

[1] Ang A, Perez A, Van Deursen A, Abreu R. Revisiting the practical use of automated software fault localization techniques. In: Int. symposium on software reliability engineering workshops. 2017, p. 175–82. [http://dx.doi.org/10.1109/ ISSREW.2017.68](http://dx.doi.org/10.1109/ISSREW.2017.68).

[2] Hirsch T, Hofer B. What we can learn from how programmers debug their code.

In: 8th Int. workshop on softw.eng. research and ind. practice. 2021, p. 37–40. <http://dx.doi.org/10.1109/SER-IP52554.2021.00014>.

[3] [Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A](http://dx.doi.org/10.1109/SER-IP52554.2021.00014) survey on software fault localization. IEEE Trans Softw Eng 2016;42(8):707–40. [http://dx.doi.org/10. 1109/TSE.2016.2521368](http://dx.doi.org/10.1109/TSE.2016.2521368).

[4] [Gazzola L, Micucci D, Ma](http://dx.doi.org/10.1109/TSE.2016.2521368)riani L. Automatic software repair: A survey. IEEE Trans Softw Eng 2019;45(1):34–67. <http://dx.doi.org/10.1109/TSE.2017.2755013>.

[5] Hirsch T, Hofer B. Root cau[se prediction based on bug reports. In: 4th](http://dx.doi.org/10.1109/TSE.2017.2755013) Int.

workshop on software faults (IWSF), ISSRE workshop proceedings. 2020, p. 171–6. <http://dx.doi.org/10.1109/ISSREW51248.2020.00067>.

[6] Lopes F[, Agnelo J, Teixeira CA, Laranjeiro N, Bernardino J. A](http://dx.doi.org/10.1109/ISSREW51248.2020.00067)utomating orthogo-nal defect classification using machine learning algorithms. Future Gener Comput Syst 2020;102:932–47. <http://dx.doi.org/10.1016/j.future.2019.09.009>.

[7] Thung F, Le X-BD, Lo [D. Active semi-supervised defect categorization](http://dx.doi.org/10.1016/j.future.2019.09.009). In: 23rd Int. conference on program comprehension. 2015, p. 60–70. [http://dx.doi.org/ 10.1109/ICPC.2015.15](http://dx.doi.org/10.1109/ICPC.2015.15).

[8] [Tan L, Liu C, Li Z,](http://dx.doi.org/10.1109/ICPC.2015.15) Wang X, Zhou Y, Zhai C. Bug characteristics in open source software. Empir Softw Eng 2014;19(6):1665–705. [http://dx.doi.org/10. 1007/s10664-013-9258-8](http://dx.doi.org/10.1007/s10664-013-9258-8).

[9] [Li Z, Tan L, Wang X,](http://dx.doi.org/10.1007/s10664-013-9258-8) Lu S, Zhou Y, Zhai C. Have things changed now?: An empirical study of bug characteristics in modern open source software.

In: 1st Workshop on architectural and system support for improving software dependability. 2006, p. 25–33. <http://dx.doi.org/10.1145/1181309.1181314>.

[10] Ray B, Posnett D, Filkov V, D[evanbu P. A large scale study of programm](http://dx.doi.org/10.1145/1181309.1181314)ing languages and code quality in GitHub. In: ACM SIGSOFT symposium on the foundations of software engineering. 2014, p. 155–65. [http://dx.doi.org/10. 1145/2635868.2635922](http://dx.doi.org/10.1145/2635868.2635922).

[11] [Ni Z, Li B, Sun X, Chen](http://dx.doi.org/10.1145/2635868.2635922) T, Tang B, Shi X. Analyzing bug fix for automatic bug cause classification. J Syst Softw 2020;163:110538. [http://dx.doi.org/10.1016/ j.jss.2020.110538](http://dx.doi.org/10.1016/j.jss.2020.110538).

[12] [Goseva-Popstojan](http://dx.doi.org/10.1016/j.jss.2020.110538)ova K, Tyo J. Identification of security related bug reports via text mining using supervised and unsupervised classification. In: Int. conf. on software quality, reliability and security. 2018, p. 344–55. [http://dx.doi.org/10. 1109/QRS.2018.00047](http://dx.doi.org/10.1109/QRS.2018.00047).

[13] [Wu X, Zheng W, Chen](http://dx.doi.org/10.1109/QRS.2018.00047) X, Zhao Y, Yu T, Mu D. Improving high-impact bug report prediction with combination of interactive machine learning and active learning.

Inf Softw Technol 2021;133:106530. [http://dx.doi.org/10.1016/J.INFSOF.2021. 106530](http://dx.doi.org/10.1016/J.INFSOF.2021.106530).

12