Array 15 (2022) 100189

Contents lists available at ScienceDirect

Array

journal homepage: 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

ARTICLE INFO

Keywords:
Bug report
Bug benchmark
Fault type prediction

ABSTRACT

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 development, 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 debugging 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 therefore 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 *Concurrency*, *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 support, 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 International 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 classifier 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 classification 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 author.
E-mail addresses: thirsch@ist.tugraz.at (T. Hirsch), bhofer@ist.tugraz.at (B. Hofer).

https://doi.org/10.1016/j.array.2022.100189
Received 8 March 2022; Received in revised 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/).

- 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 existing 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 problems. 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,[1] and sometimes incomplete bug reports, that widely vary in length,[2] content, and vocabulary.

In some cases, deriving aforementioned fault type from such bug reports can be trivial, as for this *memory* bug[3]:

> **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 error[5]:

> **ResourceLeakException in CR7** During our tests with our benchmark, we saw a Leak which according to us and norman did not make sense.

These issues make bug reports a challenging target for NLP approaches. Our task of fault type classification is further complicated by lack of information or misleading information in such bug tickets. We therefore do not expect to see the high classification performance scores known from classic NLP showcase problems.

3. Related work

Multiple researchers have investigated the application of ML and NLP methodologies on textual bug reports for classification problems. Lopes et al. [6] automatically classified more than 4000 bug reports collected from three open-source database applications according to the Orthogonal Defect Classification (ODC) schema. They performed undersampling to avoid imbalanced datasets. Thung et al. [7] performed automated ODC defect type classification using semi-supervised learning. Our work distinguishes from Lopes et al.'s and Thung et al.'s works in the used schema. While the ODC schema aims at the analysis and optimization of the software development process, our schema aims to assist developers in choosing the best debugging tool for the bug at hand.

Tan et al. [8] categorized bugs w.r.t the dimensions root cause, impact, and affected component. They applied ML classifiers as support in their data mining efforts. While Tan et al. categorize bugs into concurrency, memory and semantic bugs, we introduce a fourth category (*Other*) which covers documentation, build system, configuration and UI resource faults. Since only a small fraction of their manually labeled dataset contains concurrency bugs, they performed a keyword search using keywords such as 'race' and 'lock' on the textual bug reports. In contrast to their work, we performed the keyword search on the commit messages instead of the bug reports to reduce information leakage.

Li et al. [9] used ML classifiers to study bug characteristics in open-source software. Similar to Tan et al., they classified the root cause of bugs into Memory, Concurrency, and Semantic bugs.

Ray et al. [10] used ML classifiers to investigate programming language and code quality metrics in open-source projects. They apply five different root cause categories: Algorithmic, Concurrency, Memory, generic Programming, and Unknown. Since their approach is focused on the analysis of fixed bugs, they train their ML approach on commit messages (a posteriori approach). However, we are interested to provide information to the developers before they have fixed the bug. Therefore, we train our classifier on the textual bug report instead of the commit messages (a priori approach).

Ni et al. [11] predicted the root cause type from the code changes by converting the code changes into abstract syntax trees (ASTs) and then using a Tree-based Convolutional Neural Network (TBCNN). They distinguished six main root cause categories (function, interface, logic, computation, assignment, and others) and 21 sub-categories. In contrast to our work, this classification was performed post mortem on the bug fix.

The following approaches focus on the prediction of categories other than the root cause: Goseva et al. [12] used supervised and unsupervised learning algorithms for labeling security and non-security issues. Wu et al. [13] combined interactive ML and active learning to predict high-impact bugs. Xia et al. [14] proposed a novel feature selection algorithm for machine learning based classification of bug reports into mandelbugs and bohrbugs. Later on, Du et al. [15] proposed a cross-project transfer learning framework for the same purpose. CaPBug [16] predicts the category (e.g. Program Anomaly, GUI, Performance, Test-Code) and priority of bug reports using different classifiers, but also provides a good overview of papers for categorizing and prioritizing bugs.

ML has been used to automatically distinguish bug reports from feature requests [17], to predict the severity [18] or priority [19] of

[1] see e.g., orientdbissue2121.
[2] see e.g., elasticsearchissue4960.
[3] elasticsearchissue1118.
[4] spring-bootissue139.
[5] nettyissue1508.

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 preprocessing 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 automated 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 classification 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*. Grottke 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 software 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 understanding 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.

4.2.1. Classifiers

Multinomial Naive Bayes (MNB) are probabilistic classifiers based on Bayes theorem. Naive in this context stands for the assumption that probabilities of features are independent. MNB classifiers are fast and easy to use and can provide a performance baseline to compare other classifiers against.

Support Vector Machines (SVM) are non-probabilistic approaches that can be used for regression and binary classification. SVMs construct a hyperplane in the feature space separating the classes. For multi-class classification problems, multiple SVMs are trained simultaneously in a one-vs-all or one-vs-one setup.

Random Forrest (RF) are an ensemble learning approach that can be used for regression and classification. RF classifiers construct a series of diverse decision trees that are then combined as an ensemble to perform classification.

Logistic Regression (LR) are linear classifiers for binary classification. Logistic regression as well as linear regression are both based on linear models. LR uses the logistic (sigmoid) function to discretize their output—hence the name logistic regression. For multi-class problems, multiple LRs are combined in a one-vs-all or one-vs-one setup.

4.2.2. Evaluation strategies and balancing

K-fold Cross Validation (K-fold CV) is a statistical method for selecting parameters and evaluating ML models. The data is sliced into k equally sized parts. Training of the model is performed with $k-1$ parts and the resulting model is validated on the remaining part. This is repeated k times so that every slice is used as validation set once. Stratified k-fold CV creates slices in a manner that the original class distribution is preserved.

Bootstrap for ML classification is performed by repeatedly generating training and test splits using uniform random sampling with replacement. An item can therefore occur multiple times in a split, but it cannot occur in both test and training. The resulting performance scores are used to calculate confidence intervals for the confidence level α as the $(1-\alpha)/2$ and $\alpha+(1-\alpha)/2$ percentiles.

Imbalanced data in the context of a classification problem means the amount of items per class is not equal for all classes. While there are ML algorithms that are insensitive to imbalanced data, all of the above described classifiers are sensitive to such imbalances. There are basically two strategies to balance a training set, up-sampling and down-sampling. Up-sampling creates synthetic instances for the minority class to scale it up to match the majority classes size. Down-sampling removes instances from the majority class to scale it down to match the minority classes size.

4.3. Natural Language Processing (NLP)

In the following, we provide an overview of the preprocessing and preparation steps for applying machine learning algorithms on text documents.

In the **Tokenization** step, the input document is split into tokens. The models described in this work tokenize by words.

Vectorization creates a feature vector from a tokenized document. The models in this work utilize simple Count Vectorizers, where the feature vector represents the count of occurrences of the individual tokens in the document.

Bag of words is a common approach in which the document is split into word-tokens followed by vectorization. The order of words and their connections to each other are lost; a document is represented as an unordered collection of words.

Term frequency-inverse document frequency (Tf-idf) is a transformation method that applies weights depending on how common a term is in the whole document corpus. Tf-idf transformation of a given term t and document d is the term frequency tf, as the number of occurrences of the term in this document, times the inverse document frequency idf of the term.

$$Tf-idf(t,d) = tf(t,d) * idf(t) \quad (1)$$

$$idf(t) = log\left(\frac{n+1}{df(t)+1}\right) + 1 \quad (2)$$

The document frequency $df(t)$ is the number of documents in the corpus 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 represented 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' documentation 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 examinations 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) expresses 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 ($P = \frac{TP}{TP+FP}$). 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 ($R = \frac{TP}{TP+FN}$). 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 ($F1 = 2 * \frac{P*R}{P+R}$).

Weighted average F1 is the average of all classes' F1 scores with the number of instances of each class as weights for this class.

$$waF1 = \frac{\sum^{class} instances_{class} * F1_{class}}{\sum^{class} instances_{class}} \quad (3)$$

Macro average F1 in a multi-class problem is the unweighted mean of all classes' F1 scores. Given a balanced input, the macro and weighted average F1 scores produce the same result.

$$maF1 = \frac{\sum^{class} F1_{class}}{|class|} \quad (4)$$

Cohens kappa measures inter-rater agreement corrected for agreement by chance. It is calculated based on the proportion of items where both raters agree (p_0), and the proportion of times where agreement is expected by chance (p_c) [34]. κ values between $0.41 < \kappa < 0.60$ are considered as moderate, $0.61 < \kappa < 0.80$ as substantial, and $0.81 < \kappa < 1.00$ as almost perfect agreement [35].

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \quad (5)$$

5. Experimental setup

First, we explain the used classification schema (Section 5.1). Afterwards, we describe the underlying dataset (Section 5.2) and the creation process of the training set (Section 5.3). Finally, we present the ML modeling approach (Section 5.4).

5.1. Classification schema

Our classification schema should aid developers in the debugging process. We base it on the root cause dimension of Tan et al.'s classification schema [8]. This root cause classification is intended to encapsulate the most promising debugging approaches and specific debugging tools for each category. The top-level of our classification schema is composed of four main categories, *Concurrency*, *Memory*, *Other*, and *Semantic*. The group of useful debugging tools for *Concurrency* bugs will be disjoint from the group of tools useful for bugs in the *Other* category.

Table 1 provides an overview of our classification schema. A detailed documentation including examples and for the increasingly detailed subgroups can be found in our online appendix. The purpose of these detailed levels of our classification schema is two-fold: to serve as documentation and education tool for manual classification, and to provide additional information that infers fault and fix patterns providing a clearer picture to the reader. The top-level categories *Concurrency*, *Memory*, *Other*, and *Semantic* are used in the experiments in the next parts of this paper.

5.2. Dataset

To collect a reasonable number of issue tickets, we mined 101 open-source Java projects hosted on GitHub. These projects cover a wide variety of different software domains, ranging from server side applications, database applications, ML frameworks, testing frameworks, to mobile applications. We added all closed issue tickets of these projects whose labels contain any of the strings 'bug', 'defect', 'breaking', or 'regression' to our dataset, i.e., 54 755 issue tickets. The resulting dataset is rather noisy due to quirks in GitHub API, bug triaging performed manually by project maintainers, and varying workflows in different projects. Since GitHub API does not differentiate between issues and pull requests, the initial dataset contains both. Further, issues can be closed due to a variety of reasons including duplication, reluctance to fix, or rejected as 'not a bug' despite the label. To clean the dataset, we removed all issue tickets that

- are pull requests,
- have zero or more than 10 commits linked,

Table 1
Fault type classification schema with top-level categories in bold font and 2nd-level categories in normal font.

Fault type	Description
Concurrency	**Improper or incorrect synchronization and wrong assumptions about atomicity.**
Order violation	Incorrect order caused by missing or incorrect synchronization or thread handling.
Race condition	Two or more threads simultaneously access the same resource with at least one being a write access.
Atomic violation	Atomicity of a operation was wrongly assumed, or improper use of not thread safe data types and structures.
Deadlock	Two or more threads stuck in waiting for the other to release a resource.
Parallelization	Missing parallelization resulting in lagging, or removal of parallelization to meet constraints.
Other	Concurrency-related faults that do not fall in any of the above categories.
Memory and resources	**Improper or incorrect memory and resource handling.**
Overflow	Buffer overflows, excluding overflowing numeric types.
Null pointer dereference	Null pointer dereferences.
Uninitialized memory read	Uninitialized memory reads except null pointer dereference.
Leak	Memory and resource leaks.
Dangling pointer	Pointers referring to invalid data.
Double free	Freeing the same memory more than once.
Other	Memory-related faults that do not fall in any of the above categories.
Other	**Bugs that cannot be resolved by changing Java code.**
Documentation	Incorrect or missing documentation.
Build system	Missing or incorrect build configuration and bugs in build scripts.
UI resources	Missing or incorrect UI resources.
Configuration	Missing or incorrect config files, excluding build config and UI config.
Semantic	**Inconsistency of requirements, programmers intentions, and actual implementation that do not fall in the above categories.**
Exception handling	Incorrect, missing, or overshooting exception handling.
Missing case	Code is missing. Missing implementation, missing program flow, or missing other code parts due to unawareness of a certain case.
Processing	Code is incorrect. Incorrect implementations, ranging from miscalculations to incorrect library usage.
Typo	Simple typographic errors.
Dependency	Bugs introduced by changes in a foreign library or system that lead software that can be built, but behaves unexpected or incorrect.
Other	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 expected to be highly imbalanced w.r.t. our classification target. Other researchers 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 downsampling 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 carry an understanding of the underlying problem, while also carrying a certain level of technical expertise and associated vocabulary. Further, the corpus of commit messages is distinct from the corpus of textual bug reports that constitutes the inputs of our models. This reduces unwanted biases towards certain keywords in our machine learning experiments.

Selection of bug reports and manual classification. We split our dataset into two groups: issues containing Java code changes (10 146 issues) and issues that do not (1 475 issues). On this first group, we applied our keyword search, yielding 756 results, of which we manually analyzed 514. Further, we randomly selected and analyzed 158 issues from this group to adequately sample the majority class (*Semantic*). From the second group, we randomly selected and analyzed 508 issues to adequately sample our *Other* category. In total, we manually analyzed 1180 issues from 86 software projects. During this manual examination efforts, we removed 418 issues from the dataset for the following reasons:

- Reports that were not considered a bug.
- Corresponding fixes were in programming languages other than Java, if they did not fall into the *Other* category.
- Non-English bug reports
- The fault type was indeterminable within reasonable amount of time.

The most common reason for an indeterminable fault type were refactorings, enhancements, and other major code changes within the linked commits that make it impossible to isolate the code responsible for the bug without in-depth project knowledge.

The resulting data set has a total size of 762 bug reports from 75 different software projects. It contains 155 *concurrency*, 132 *memory*, 142 *other*, and 333 *semantic* bugs.

Dataset quality and verification. Researcher 1 performed the manual classification described above. Six months later, Researcher 1 reclassified 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 classification. 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 undersampling the majority classes, resulting in equally sized categories containing each 124 bug reports. The final data set used in our experiments 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), Multinomial Naive Bayes (MNB), Random Forrest (RF), and Support Vector Machines (SVM) based on other researchers' work in similar endeavors [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, decamelcasing, 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 $\alpha = 0.95$. 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.

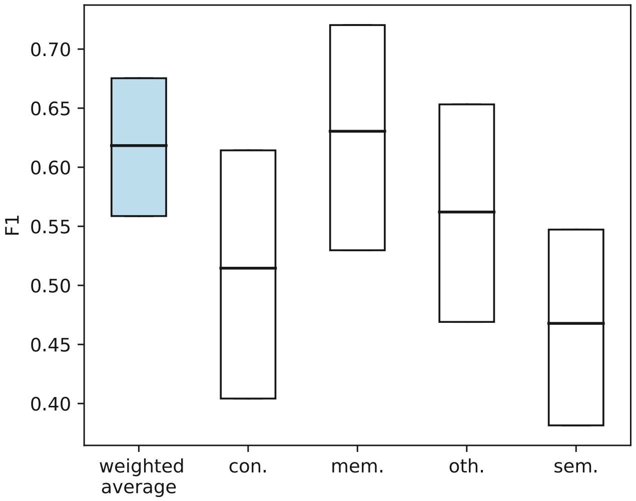


Fig. 1. Survey F1 Bootstrap confidence intervals.

Approach: We performed an online user survey tasking participants to classify textual bug reports according to their fault type in four categories. The survey was promoted via email and direct messages to professional developers at resident software companies and to master students in the field of computer science. It can be assumed that the majority of survey participants were students. Anonymous participation was allowed, while participants disclosing a contact e-mail address could win vouchers of a well-known online store. Multiple submissions from the same person were allowed. For each submission, we tracked IP address, email address (optional), time spent on each bug report, and the corresponding answers for each bug report.

A balanced set of 500 bug reports was randomly sampled from the 762 bug reports already investigated by Researcher 1 (see 5.3). Participants were provided ten bug reports randomly selected from a pool of 500. The bug reports were formatted the same way as the originals on GitHub, and single choice answers for each bug report were available. While we did not provide a link to the original bug ticket on GitHub, inline images as for example, screenshots, and links in the bug report texts were preserved. Participants were given a short introduction into our classification schema on the survey's entry page. Short explanations of the four fault types were available during participation inline of each page. Further, a link to detailed explanations and examples of our classification schema were provided. There was no time limit.

Results: We received 51 submissions. We carefully investigated the times spent and IP addresses and did not find any submissions indicative of being made with malicious intent. The submissions provide us with a total of 510 manually classified bug reports for our analysis. Removal of bug reports originating from three projects as discussed in , leaves us with 483 classified bugs as the basis for the following discussion.

Fig. 1 shows the Bootstrap intervals from survey submissions ($n = 1000$, $\alpha = 0.95$). The mean performance in terms of weighted average F1 over all classified bug reports is 0.62. Participants performed best on *Memory* bugs, with a mean F1 score of 0.63, and worst for *Semantic* bugs with a mean F1 score of 0.47.

Fig. 2 shows the normalized confusion matrix of all received answers. *Other* bugs are most frequently misclassified as *Semantic* bugs by participants. Recall is lowest for *Concurrency* bugs (0.56) and highest for *Memory* bugs (0.68). Precision is lowest for *Semantic* bugs (0.48) and highest for *Other* bugs (0.70).

Discussion: Developers foreign to a software project perform with a weighted average F1 score of 0.62, and can correctly identify the fault type based on a bug report in 62 % of the cases. As we lack the sample size to investigate reasons for misclassification on a single bug

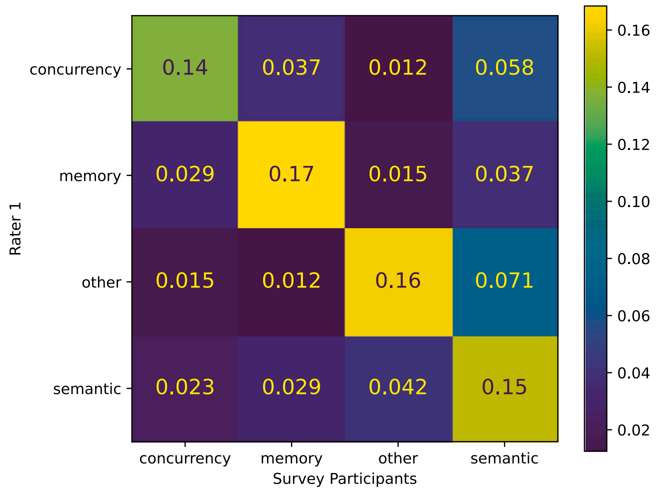


Fig. 2. Confusion matrix of all received answers.

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 1508[6] for 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 classifying bug reports?

Motivation: We investigate the performance of classic ML approaches for fault type classification. Despite this narrow focus, capturing 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 increased performance for each bug class as can be observed in Fig. 3(b). While for example, the model with artifact removal performs significantly 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.

Results: Fig. 4(a) shows the macro average F1 Bootstrap confidence intervals and mean performance of the resulting models. The mean macro average F1 scores of the classifiers are closely clustered, 0.64 for MNB, 0.65 for SVM, 0.65 for RF, and 0.66 for LR. Only LR scores are statistically significant ($p < 0.025$) different from the other models' scores ($p = 0.003$).

However, investigating the models' performance on specific classes shows that despite their similar macro average F1 scores, there are differences in their capabilities. Fig. 4(b) shows the F1 Bootstrap confidence intervals and mean performance of the models for each class. The most apparent difference can be observed in the performance of the LR and RF models on *Memory* and *Other* bug classes. The mean F1 performance of the LR and RF models are 0.69 and 0.65 for the *Other* class, and 0.74 and 0.79 for the *Memory* class. One sided T-tests on the models' performance scores show that the RF model is statistically significant better than LR model for *Memory* bugs ($p = 2 * 10^{-10}$), and vice versa for *Other* bugs ($p = 2 * 10^{-10}$).

EXP3: To leverage the insights from EXP1 and EXP2, we use ensemble methods. To train and identify the best models for each class, we perform nested cross validation for each class while adding weights to this class and at the same time using the class specific F1 measure as metric for inner CV hyperparameter tuning. We perform this step for all classifier algorithms. Given the n fold outer CV, this provides us with $n * numClassifierAlgorithms$ models for each class. From these, we select the best m models for each class and ensemble them using a stacking classifier ensemble with LR as base classifier. The base LR classifiers' parameters are kept at the default values. The resulting ensemble size is $m * numClasses$. We trained and evaluated the resulting ensembles with $m = 1, 2, 5$ using Bootstrap.

Results: The models contained in the ensembles and their corresponding hyperparameters are shown in Table 2, Ensemble 1 consisting of the topmost four classifiers, Ensemble 2 of the topmost eight classifiers, and Ensemble 3 of all classifiers listed in the table. Due to aforementioned selection process we observe certain hyperparameters to be connected to specific target classes, most notably artifact removal as already discussed in EXP1.

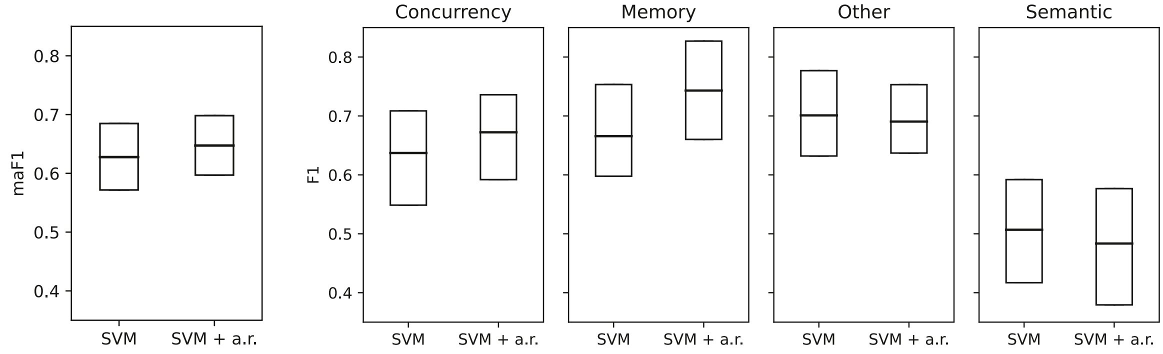
Fig. 5(a) shows the macro average F1 Bootstrap confidence intervals and mean performance of the resulting ensembles and the on average best performing models from EXP2. The mean macro average F1 performances of the ensembles are closely clustered, ranging from 0.68 for Ensemble 1 to 0.69 for Ensemble 3. T-test on the ensembles' scores shows that we cannot reject the null hypothesis that the distributions of scores differ ($p = 0.118$ for Ensemble 1 scores to be different from Ensemble 2 scores, and $p = 0.078$ for Ensemble 1 scores to be different from Ensemble 5 scores).

However, Ensemble 1 with a mean macro F1 score of 0.68 performs significantly better than the single classifier models from EXP2. Results for one sided T-test for Ensemble 1 scores being greater than the LR classifier from Experiment 2 are $p = 4 * 10^{-11}$ (Ensemble 1 > SVM: $p = 4 * 10^{-19}$).

Fig. 5(b) shows the ensembles' performance across different bug classes. One sided T-testing Ensemble 1 scores against LR show that it performs significantly better for the class of *Other* ($p = 0.008$) and *Semantic* ($p = 1 * 10^{-10}$) bugs. The same test for the best classifiers for *Memory* and *Concurrency* bug classes from EXP2 (RF and LR respectively) does not allow to reject the null hypothesis that Ensemble 1 performs better (RF *Memory* $p = 0.060$ and LR *Concurrency* $p = 0.069$). However, these results suggest that Ensemble 1 performs at least equally well for these bug classes.

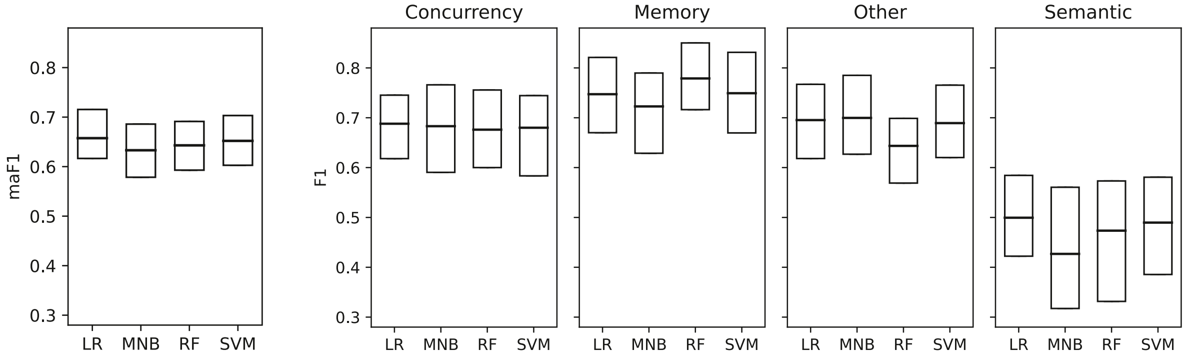
Fig. 6 shows the cumulative confusion matrix of Ensemble 1 collected from the 100 Bootstrap iterations. *Semantic* bugs are the ones most often confused with other categories, most notably the *Other* category. To investigate the reasons of misclassifications, we collected all test sets and their predictions from all Bootstrap iterations of Ensemble 1, yielding 22286 bug reports and their predictions. 7072 of which

[6] nettyissue1508.



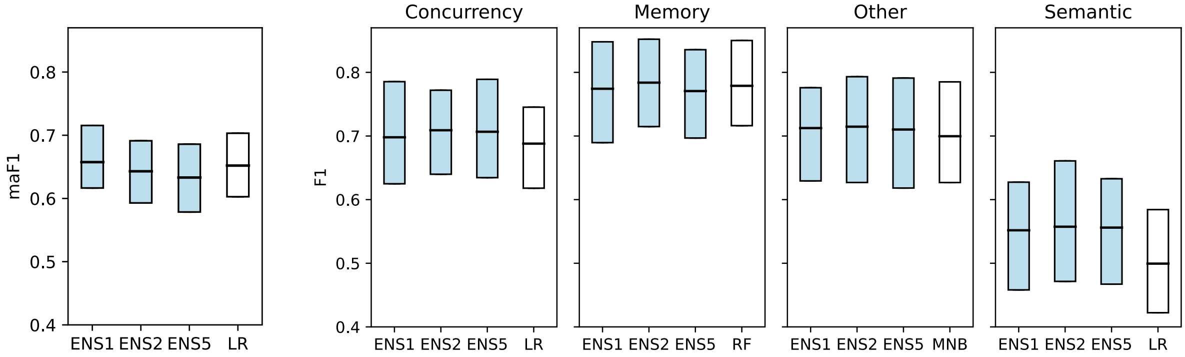
(a) Macro average F1 (b) Single class F1

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



(a) Macro average F1 (b) Single class F1

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



(a) Macro average F1 (b) Single class F1

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 artifact portions in the always correctly predicted group, the sample size is too small and too noisy to allow any conclusions to be drawn.

Therefore, we investigated the content of these bug tickets rather than statistical measures on their size. We created a vocabulary from all words included in these bug reports, using case folding and stop word removal, as well as removal of words containing digits and special characters. We counted in how many bug reports each word occurs. Fig. 7 shows the top twenty words for both groups. For the group of always correctly identified items, the corresponding bug reports are explicit especially regarding *Memory* and *Concurrency* issues, while the always incorrectly identified items are lacking such expressiveness.

We further investigated the fault types of the always correct and always incorrect groups of bug tickets. The latter group is headed by *Semantic* bugs (11) while the always correctly identified group is strongly dominated by *Memory* bugs (56).

Table 2
Hyperparameters used in ensemble classifiers.

			Classifier	Target	Artifact removal	Class weight	Decamel-case	Tf-idf	Stemm-ing	Stop word removal
Ensemble 3	Ensemble 2	Ensem. 1	SVM	Concurrency	true	0_4	false	true	true	false
			RF	Memory	true	1_4	true	true	false	English
			MNB	Other	false	–	false	true	false	English
			RF	Semantic	false	3_4	true	true	true	false
			RF	Concurrency	false	0_4	true	true	true	English
			MNB	Memory	true	–	false	false	true	English
			MNB	Other	true	–	false	true	true	English
			MNB	Semantic	false	–	false	false	true	false
			MNB	Concurrency	true	–	false	false	true	English
			MNB	Concurrency	true	–	false	false	true	English
			MNB	Concurrency	true	–	false	false	true	English
			SVM	Memory	true	1_4	false	true	true	false
			RF	Memory	true	1_4	true	true	false	English
			RF	Memory	true	1_4	true	true	true	English
			MNB	Other	true	–	true	true	true	false
			SVM	Other	false	2_4	true	true	true	false
			SVM	Other	false	2_4	true	true	true	false
			SVM	Semantic	false	3_4	true	true	true	false
			RF	Semantic	false	3_4	true	true	true	false
			RF	Semantic	false	3_4	true	true	true	English

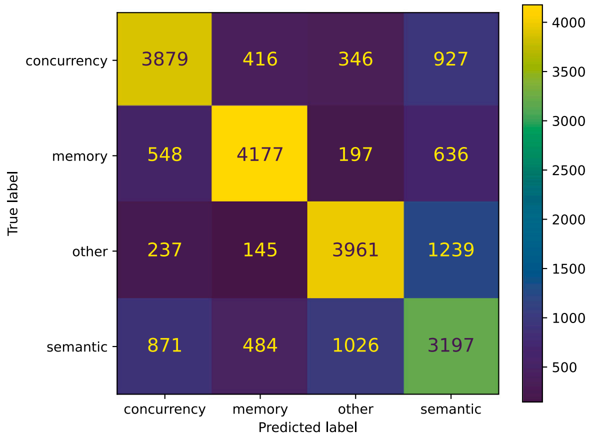


Fig. 6. EXP3: Cumulative confusion matrix from 100 Bootstrap iterations for Ensemble 1.

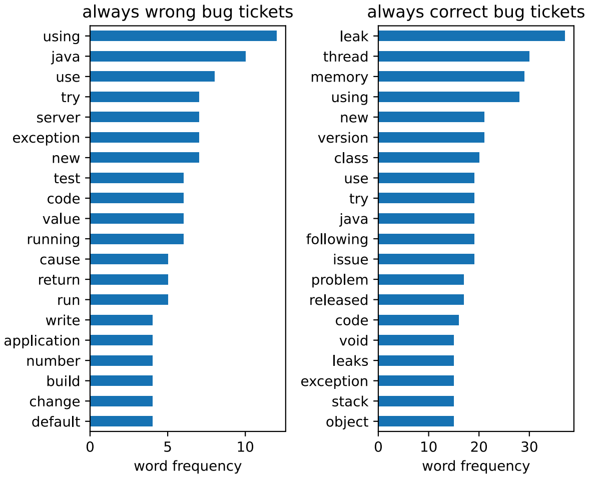


Fig. 7. EXP3: Words contained in always incorrectly and always correctly predicted bug reports with Ensemble 1.

Discussion: Our results show that preprocessing steps as artifact removal, although increasing overall performance, have a detrimental effect on certain categories. Further, we found that different classifier algorithms differ significantly in their capabilities to correctly identify certain classes.

We show that both findings can be exploited by ensemble methods to deliver better overall classification performance without the drawbacks of separate classifiers and a singular preprocessing pipeline. Our results suggest that ensemble size is not a critical factor, and an ensemble consisting of one classifier fine tuned to a specific class already provides significantly better performance than any single classifier approach.

Our investigation into this smallest ensemble's classification capabilities shows that *Semantic* bugs are the hardest to identify correctly, and that they are most often confused with *Other* bugs.

Comparing misclassified bug reports against correctly classified ones does not show any statistical relevant difference in terms of document length or contents in terms of exception names or artifacts. However, investigation into the smaller subgroups of always incorrectly or always correctly identified items shows the latter group's higher expressiveness in terms of vocabulary that can directly point towards the bug's fault type. As expected, *Semantic* bugs dominate the always incorrectly identified items, while *Memory* bugs are the most frequent type in the always correctly identified items.

Answer: We achieve a mean macro average F1 score of 0.68, and a Bootstrap confidence interval ($\alpha = 0.95$) ranging from 0.63 to 0.73 when using ensemble methods. The ensembles perform significantly better than single classifier models (MNB, LR, RF, SVM, scoring 0.66, 0.65, 0.64, 0.65 mean macro average F1 respectively). The ensemble size does not significantly increase performance in our experiments.

6.3. *RQ3: Can the proposed ML approach be used for inter-project classification?*

Motivation: Here, we consider the practicability, portability, and limitations of the approach.

Approach: We measure the performance of our machine learning approach in the scenario that training sets and validation sets are build from different software projects. We generated 100 validation and training splits by random selection of origin software projects from our dataset. These splits were chosen so that validation split size is between 18% and 20% (90–98 bug reports) of the full dataset and validation splits are roughly stratified (number of bug reports for each bug class within 0.5 of the mean of all four classes). The resulting test training splits are comprised of issue tickets originating from different software projects.

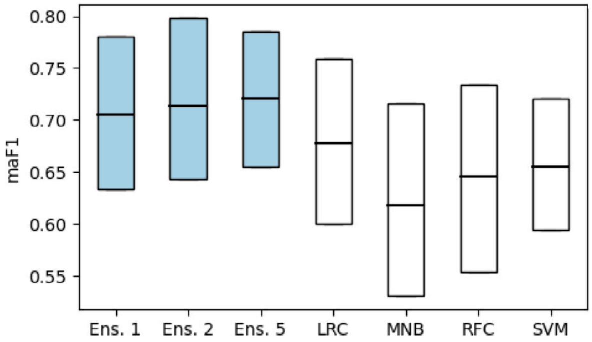


Fig. 8. 95% percentiles of F1 macro average scores for 100 project splits.

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 $p = 1 * 10^{-6}$ and $p = 2 * 10^{-18}$.

We calculate the project specific F1 score from the collective test sets and their predictions from all iterations where this project occurred. 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]

Analog to our analysis in RQ2, we investigated the bug tickets that were either always correctly classified (99 bugs), or always incorrectly classified (24 bugs). The always correct classified bug reports are headed by the *Other* class (36 bugs), followed by *Memory* (30 bugs), *Concurrency* (29 bugs), and *Semantic* (4 bugs) classes. The always incorrectly classified bug reports are headed by the *Semantic* class (10 bugs), followed by *Memory* and *Concurrency* classes (each 5 bugs), and *Other* class (4 bugs). Manual examination of the always misclassified bugs revealed 6 bugs that we consider hard to classify and 6 that we consider impossible to classify with the given information; the remainder provide enough information and context. The impossible to classify bugs are e.g., reports on improper function that arose from wrong or improper documentation and were fixed as documentation corrections or enhancements, or reports on incorrect function where the root cause was missing UI resource files.

Discussion: A comparison with our findings from RQ2 shows that the mean classifiers performance is higher for cross project application of our model. To investigate this effect, we evaluate the classifier performance on a single project level to study their impact on overall performance results.

The hypothesis that the occurrence of *elasticSearch* in so many splits inflates overall performance can be refuted given the low performance on its bug reports. Given the wide range of classification performances and occurrences of projects in our splits we cannot identify a single cause in form of a specific software project that would overly inflate our model's performance measures.

However, besides the discussed test splits, there is another significant difference to our RQ2 experiments regarding the training splits: bug reports from *netty* are always in the training sets. It is possible that *netty* bug reports are important to the training process. Unfortunately, we cannot remove *netty* from the training set while maintaining balance, because of its significant size of 84 items (including 40 memory bugs).

Answer: Examination of the classifiers' performance on individual software projects shows a wide spread of performance scores, with certain projects performing extremely well (e.g., *spring-framework*), and others rather poor (e.g., *n4js*). However, our analysis and the small available sample sizes do not allow an a priori prediction of a project's suitability for our approach.

The mean performance on 100 validation splits shows that the models are portable and robust, and that the performance results from RQ2 can be maintained in a cross project application. Our smallest ensemble models mean macro average F1 performance from all 100 validation splits is 0.71, our biggest models score is 0.72. Analog to our results from RQ2, these ensembles perform uniformly better than single classifier models (MNB, LR, RF, SVM, scoring 0.68, 0.65, 0.62, 0.65 mean macro average F1 respectively).

7. Threats to validity

The biggest threat to validity is the small sample size. To attenuate effects arising from such a small dataset, we repeatedly ran our experiments in a randomized fashion and performed statistical tests on our results.

Further, hidden biases may reside in bug reports from any single software project. To counter this threat, we sourced this dataset from 71 different open source Java projects covering various organizations, software domains, and deployment targets, ranging from search engines and database applications to small Java libraries and mobile applications. To validate our results and to demonstrate the transferability of our approach, we performed our experiments on test/training splits along software projects, ensuring that bug reports in the test set are from different software projects than the bug reports used in training.

A threat to external validity is that our dataset contains only Java bugs. In addition, only open source projects hosted on GitHub were considered in this work. We can therefore not generalize our findings

[7] https://github.com/spring-projects/spring-framework/issues/24265 vs. https://github.com/eclipse/n4js/issues/444.

Table 3
Ensemble 1 performance on specific projects bugs in cross project evaluation.

Project	Bug reports	Bug reports occurred	Splits containing project	Precision (w.a.)	Recall (w.a.)	F1 (w.a.)
spring-framework	11	165	15	0.94	0.93	0.93
spring-security	20	180	9	0.86	0.85	0.85
redisson	25	1325	53	0.88	0.83	0.85
junit5	12	456	38	0.93	0.82	0.85
ExoPlayer	12	324	27	0.84	0.83	0.84
spring-session	15	680	40	0.79	0.78	0.78
hazelcast	16	448	28	0.76	0.71	0.72
async-http-client	14	448	32	0.79	0.69	0.71
ALL_OTHERS	185	1349	100	0.68	0.66	0.67
elasticsearch	41	3198	78	0.67	0.65	0.65
checkstyle	14	196	14	0.72	0.67	0.63
spring-boot	35	455	13	0.72	0.58	0.58
n4js	12	264	22	0.60	0.59	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., feature 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 students in computer science. These participants are to be considered non-experts, as they are not involved in the development of the software 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 approaches 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 humans (0.63 mean weighted average F1 score), as well as standalone classifier models (0.79 mean macro average F1 for Random Forrest classifiers), 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).

We investigated transferability of our models by applying them on bug reports originating from software projects other than the software projects used in the training set. Transfer worked extremely well on some projects (e.g., spring framework with a macro average F1 score of 0.93), while performing poorly on others (e.g., N4js with a macro average F1 score of 0.55). However, mean performance scores drawn from 100 of such transfer experiments were consistent with our earlier findings (e.g., mean macro average F1 score of 0.71 for our smallest ensemble), support that our approach provides transferable models.

Our results show that predicting the correct bug type solely on an initial bug report is difficult. The reasons for this are 1. the noisy nature of bug reports, containing various types of non-human language artifacts, 2. the extreme variety in resulting length and structures, or lack thereof, 3. the different view points and scopes of their reporters, e.g., an end user describing the impact of the bug in non-technical terms, or a senior developer delegating work to other developers—describing the problem in technical detail including the bug's location and what actions have to be taken to fix the bug.

Our classification schema aims to provide a high level abstraction of debugging approaches and tools necessary for effective debugging of these fault types. Using this schema, we attempt to encapsulate a small portion of senior developer's knowledge in machine learning models through learning from historical bug reports and their corresponding fixes. In this work, we identify the issues arising in such endeavor, and propose and demonstrate possible solutions for some of these issues.

While our results are promising when compared to our human classifier performance baseline, application of our models as debugging support in a production environment scarcely warranted at this stage and future research in this direction is needed. However, our approach in its current form can support researchers in their effort of creating bug benchmarks of specific bug types for experiments on specialized debugging tools. Our datasets and implementations are made publicly available on GitHub.[8] and Zenodo[9] We hope that other researchers benefit from our detailed investigation and the provided artifacts.

CRediT authorship contribution statement

Thomas Hirsch: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Birgit Hofer:** Conceptualization, Validation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition.

[8] https://github.com/AmadeusBugProject/fault_type_prediction.

[9] https://doi.org/10.5281/zenodo.6539173.

Declaration of competing interest

The authors declare that they have no known competing financial 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.

[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 survey on software fault localization. IEEE Trans Softw Eng 2016;42(8):707–40. http://dx.doi.org/10.1109/TSE.2016.2521368.

[4] Gazzola L, Micucci D, Mariani 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 cause prediction based on bug reports. In: 4th 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. Automating orthogonal 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. In: 23rd Int. conference on program comprehension. 2015, p. 60–70. http://dx.doi.org/10.1109/ICPC.2015.15.

[8] Tan L, Liu C, Li Z, 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.

[9] Li Z, Tan L, Wang X, 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, Devanbu P. A large scale study of programming 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.

[11] Ni Z, Li B, Sun X, Chen 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.

[12] Goseva-Popstojanova 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.

[13] Wu X, Zheng W, Chen 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.

[14] Xia X, Lo D, Wang X, Zhou B. Automatic defect categorization based on fault triggering conditions. In: Int. conference on engineering of complex computer systems. 2014, p. 39–48. http://dx.doi.org/10.1109/ICECCS.2014.14.

[15] Du X, Zhou Z, Yin B, Xiao G. Cross-project bug type prediction based on transfer learning. Softw Qual J 2020;28(1):39–57. http://dx.doi.org/10.1007/S11219-019-09467-0/FIGURES/5.

[16] Ahmed HA, Bawany NZ, Shamsi JA. Capbug-A framework for automatic bug categorization and prioritization using NLP and machine learning algorithms. IEEE Access 2021;9:50496–512. http://dx.doi.org/10.1109/ACCESS.2021.3069248.

[17] Pandey N, Sanyal DK, Hudait A, Sen A. Automated classification of software issue reports using machine learning techniques: an empirical study. Innov Syst Softw Eng 2017;13(4):279–97. http://dx.doi.org/10.1007/s11334-017-0294-1.

[18] Alharthi ZSM, Rastogi R. An efficient classification of secure and non-secure bug report material using machine learning method for cyber security. Mater Today 2021;37(Part 2):2507–12. http://dx.doi.org/10.1016/J.MATPR.2020.08.311.

[19] Ortu M, Destefanis G, Swift S, Marchesi M. Measuring high and low priority defects on traditional and mobile open source software. In: 7th Int. workshop on emerging trends in software metrics. 2016, p. 1–7. http://dx.doi.org/10.1145/2897695.2897696.

[20] Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Comput Surv 2018;51(4). http://dx.doi.org/10.1145/3212695.

[21] Yang Y, Xia X, Lo D, Grundy J. A survey on deep learning for software engineering. 2020, arXiv. arXiv:2011.14597.

[22] Fang F, Wu J, Li Y, Ye X, Aljedaani W, Mkaouer MW. On the classification of bug reports to improve bug localization. Soft Comput 2021;25(11):7307–23. http://dx.doi.org/10.1007/S00500-021-05689-2/TABLES/6.

[23] Huang Q, Xia X, Lo D, Murphy GC. Automating intention mining. IEEE Trans Softw Eng 2020;46(10):1098–119. http://dx.doi.org/10.1109/TSE.2018.2876340.

[24] Ploski J, Rohr M, Schwenkenberg P, Hasselbring W. Research issues in software fault categorization. ACM SIGSOFT Softw Eng Notes 2007;32(6):6. http://dx.doi.org/10.1145/1317471.1317478.

[25] Endres A. An analysis of errors and their causes in system programs. In: Int. conference on reliable software. 1975, p. 327–36. http://dx.doi.org/10.1145/800027.808455.

[26] Gray J. Why do computers stop and what can be done about it? Technical report 85.7, 1985, URL http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

[27] Grottke M, Trivedi K. A classification of software faults. In: 16th Int. symp. on software reliability engineering (ISSRE'05) - sup. proceedings. 2005, p. 19–20.

[28] Chillarege R, Bhand ari IS, Chaar JK, Halliday MJ, Ray BK, Moebus DS. Orthogonal defect classification: A concept for in-process measurements. IEEE Trans Softw Eng 1992;18(11):943–56. http://dx.doi.org/10.1109/32.177364.

[29] IEEE. Std 1044-2009 - IEEE stand ard classification for software anomalies. 2010.

[30] Baeza-Yates Berthier Ribeiro-Neto R. Modern information retrieval the concepts and technology behind search. second ed.. Addison Wesley; 2011, p. 68–74.

[31] Scikit-learn documentation. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature{_}extraction.text.TfidfTransformer.html.

[32] Calefato F, Lanubile F, Vasilescu B. A large-scale, in-depth analysis of developers' personalities in the apache ecosystem. Inf Softw Technol 2019;114:1–20. http://dx.doi.org/10.1016/J.INFSOF.2019.05.012.

[33] Hirsch T, Hofer B. Identifying non-natural language artifacts in bug reports. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). 2021, p. 191–7. http://dx.doi.org/10.1109/ASEW52652.2021.00046.

[34] Cohen J. A coefficient of agreement for nominal scales. Educ Psychol Meas 1960;20(1):37–46. http://dx.doi.org/10.1177/001316446002000104.

[35] Landis JR, Koch GG. The measurement of observer agreement for categorical data. Biometrics 1977;33(1):159. http://dx.doi.org/10.2307/2529310.

[36] Thung F, Lo D, Jiang L. Automatic defect categorization. In: Work. conf. on reverse engineering. 2012, p. 205–14. http://dx.doi.org/10.1109/WCRE.2012.30.