This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

Tracking Buggy Files: New Efficient Adaptive Bug Localization Algorithm

Mikołaj Fejzer, Jakub Narębski, Piotr Przymus, and Krzysztof Stencel

Abstract—Upon receiving a new bug report, developers need to find its cause in the source code. Bug localization can be helped by a tool that ranks all source files according to how likely they include the bug. This problem was thoroughly examined by numerous scientists. We introduce a novel adaptive bug localization algorithm. The concept behind it is based on new feature weighting approaches and an adaptive selection algorithm utilizing pointwise learn–to–rank method. The algorithm is evaluated on publicly available datasets, and is competitive in terms of accuracy and required computational resources compared to state–of–the–art. Additionally, to improve reproducibility we provide extended datasets that include computed features and partial steps, and we also provide the source code.

Index Terms—Bug reports, software maintenance, learning to rank

1 Introduction

Software defects or bugs occur in the development cycle of most software projects and can cause severe problems [1]. Thus, more than one third of the costs associated with software development is spent on finding and fixing bugs [2]. One of the main sources of information about software bugs is delivered by users of said software, who submit *bug reports* containing details about encountered defects. The goal of project maintainers is then to triage, reproduce, find causes and fix reported bugs, based on provided information. To fix the bug the developer has to find the cause and *relevant* files that need to be changed; such process is called *bug localization*. Finding relevant files may be a non trivial task as the quality of bug reports will vary depending on a user's technical knowledge. As a result such reports might be incomplete, or miss some crucial information. Furthermore, usually only a few files require fixing, with the median of 3 fixed files per bug report [3], but the source code repository may contain thousands of unaffected files. Therefore, a deep understanding of the project structure and the familiarity with the relevant source code is crucial in the bug localization process [4].

Bug localization can be simplified by automatically suggesting which files require fixing, which was shown in numerous papers [5], [6], [7], [8], [9], [10], [11], [12], [13]. Automated bug localization can significantly decrease the time spent by developers and reduce the costs of software construction and maintenance. It can be seen as a specific form of *information retrieval (IR)* process, if we treat a bug report as a form of query, and a software repository as a collection of documents. As a result, files are ranked according to their relevance, that is the likelihood of containing faulty code in the context of the given bug report. Furthermore, thanks to rich software development infrastructure, information about bugs and source code can be obtained from multiple structured sources like bug reports and the change history of the source code. Bug reports are usually stored in a separate *bug tracking system*. The history of changed files can be obtained from dedicated *version control system*, containing the authorship information, the timestamps and the description of source code changes (*commits*). Both software repositories and bug trackers can be treated as rich data sets for information retrieval purposes [14].

There are some challenges in the bug localization field. First, there exists a difference between the natural language used in bug reports and the programming language employed in the source code. Thus, using simple lexical matching scores may result in suboptimal performance. For that reason a set of specialised, domain knowledge based features is often used [3], [12]. Second, there is a large disproportion between the number of relevant (buggy) files and the rest of the project. Hence, both model and training data need to be carefully selected. One of common pitfalls is the imbalance of positive examples and false positives from closely related files (e.g. files from the stack trace).

Numerous algorithms have been proposed for bug localization, with some of the earliest models checking presence of API calls in stack traces reported [5] or by computing code metrics and finding outliers [6], [7]. Current bug localization algorithms take advantage of information retrieval and machine learning algorithms, utilizing various features present in software repositories and bug report systems. The text data obtained from both reports and source code can be used to find similarities between bugs and files [8], [9]. Abstract Syntax Tree (AST) may be used to extract more fine-grained information, like names of classes, methods, variables, and source code comments; which can further improve bug localization [10]. This process can be additionally enhanced by extracting stack traces from bug reports [11]. More complex systems use a composition of existing algorithms, by using linear combinations of ranking scores [15], [16], [17] or by using learning to rank algorithms [12], [13].

Notable results, including a set of features and a new learn-to rank algorithm, were presented by Ye et al. [12],

- *M. Fejzer, J. Narębski and P. Przymus are with the Nicolaus Copernicus University in Toruń*
  E-mail: {mfejzer, jnareb, piotr.przymus}@mat.umk.pl
- *K. Stencel is with the University of Warsaw, Poland.*
  E-mail: stencel@mimuw.edu.pl

outperforming many others. Moreover, authors proposed a new fine-grained dataset that better reflects practical applications than commonly used datasets [9]. Both the algorithm and the dataset were an important step in the bug localization research field.

Automatic bug localization is practically (better code quality) and scientifically (new interesting algorithms) important. Thus, we propose a new algorithm based on pointwise learn-to-rank. For all used metrics and evaluated projects we improve or stay comparable with other state–of–the–art algorithms. Most notably, we always improve Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and Accuracy@1. Furthermore, our algorithm consumes significantly less computing resources and is fully adaptive. No tuning of the parameters (e.g. weights) is required. In contrast such a step is executed separately in other approaches. Additionally, we provide source code of our algorithm[1] to simplify replication.

This paper is structured as follows. In Section 2 we discuss related work. In Section 3 we define the problem, describe the features and present our algorithms. In Section 4 we present the datasets, report the experimental results and discuss our findings. In Section 5 we address threats to validity. In Section 6 we conclude the paper and suggest possible future work.

2 Related work

The problem of bug localization has been thoroughly examined by numerous scientists (see Table 1 and in Fig. 1). In this section we split the related work according to the common features and discuss them in four groups of similar approaches.

Algorithms based on augmented bugreport–source similarity use a single feature or combine different features without an explicit method to learn the weights applied to compute the final scoring. The key problem with such techniques is the lack of generality, as they tend to overfit specific datasets. The learn-to-rank approaches, such as our proposed algorithm, do not suffer from manually selected weights.

Nguyen et al. [8] prepared a bug file localization tool called BugScout, based on a modified Latent Dirichlet Allocation (LDA) topic modelling algorithm and Gibbs Sampling method. Two topic models were prepared, respectively for bug reports and source code files. Links between reports and fixes were used to connect topic distributions in both models. The cosine distance was used as the similarity measure between topics, and together with defect-proneness factor, used to rank files.

BugLocator [9] is a bug localization algorithm based on the textual similarity between a bug report and the source code. Zhou et al. proposed a modification to the Vector Space Model (VSM) with changes to TF-IDF function taking into account that larger source files contain on average more bugs and should not be penalized. Additionally, the algorithm utilizes the similarity score (SimiScore) defined between a given bug and recently closed bugs to adjust the file rankings.

The authors of BLUiR [10] proposed an algorithm that uses features obtained via AST parsing, like class names, method names, variable names and comments. To measure the similarity between a bug report and a file, the approach uses the Indri TF-IDF model and incorporates structural IR.

BRTracer [11] utilizes a segmentation of source code files and stack traces to reduce the noise from large files. Each source file is converted into segments and the segment with most resemblance to the bug report is used in further analysis. The algorithm calculates the similarity between segments and bug reports, with an additional boost score calculated from files present on the stack trace.

AmaLgam [15] integrates BugLocator [9] and BLUiR [10] with repository data. This algorithm combines other algorithms by using a weighted sum, with weights determined experimentally per each project. It suggests relevant buggy source files by combining BugLocator's SimiScore and BLUiR's structured retrieval into a single score using a weight factor. This is then combined with a version history score that represents the number of bug fixing commits for given file in the past k days.

The BLIA [17] tool localizes bugs on the levels of a file and of a method. The authors utilized the revision history, file contents, bug reports with comments and stack traces to find suspicious files. All methods in suspicious files are analysed in terms of the similarity to bug reports. The similarity score between a file and a report is based on BLUiR and the total score is based on AmaLgam [15].

ConCodeSe [20] aims at improving the bug localization without features extracted from a project history and is based on a probability and a lexical scores from Apache Lucene.

Algorithms based on learn-to-rank use machine learning to prepare ranking model from multiple features. In such bug localization algorithms published so far the parameters/weights are selected globally. They do not adapt to, e.g. changing development practices. However, in our proposed algorithm we incorporate the parameter adaptation into the main training loop without any performance loss.

Ye et al. proposed to detect defects by Learning to Rank [3]. The algorithm uses text similarity measures on source code, utilizing an enriched API hierarchy, class name similarity, collaborative filtering, bug fixing recency and frequency. A learning to rank classifier was provided by the SVMrank package. This algorithm, further improved in the following Ye et al. publication [12], utilizes additional features obtained from AST parsing similarly, and retrieved from the class dependency graph. Both Ye et al. [3], [12] algorithms use SVMrank, an example of *pairwise approach*.

An extended AmaLgam+ [16] algorithm was proposed by authors of Amalgam [15], which introduced genetic algorithm to learn weights, based on *listwise* principle. Additional features were added based on stack trace analysis via BRTracer [11].

Shi et al. [13] proposed using learning to rank approaches with various algorithms implemented in RankLib [22]. The Random Forests and MART algorithms operate on *pointwise* principle. For *pairwise algorithms* Shi et al. tested RankNet, RankBoost and LambdaMART. The CoordinateAscent (selected) is the *listwise* example. The features used were similarity scores between bug report descriptions with class names, methods, variables and comments, stack traces, version history per each file, dependence graphs, and report-to-report similarities.

1. https://github.com/mfejzer/tracking_buggy_files

TABLE 1
A summary of the related work in bug location problem area, and datasets used. Top N and Accuracy@k metrics are convertible. Here rVSM – revised Vector Space Model, LtR – Learning to Rank, DNN – Deep Neural Network, RBM – Restricted Boltzmann Machine.

Name	Date	Dataset	Approach	Metrics
BugScout [8]	2011	ArgoUML, AspectJ, Eclipse, Jazz	LDA w. Gibbs sampl.	accuracy, Top 10
BugLocator [9]	2012	AspectJ, Eclipse, SWT, ZXing	rVSM + SimiScore	Top N, MAP, MRR
BLUiR [10]	2013	*BugLocator dataset [9]*	struct. rVSM: Indri toolkit	Top N, MAP, MRR
two-phase [8], [18]	2013	8 modules from Firefox and Core in Mozilla	Naïve Bayes	accuracy, precision, recall
BRTracer [11]	2014	AspectJ, Eclipse, SWT *(from BugLocator [9])*	rVSM + stack trace info	Top N, MAP, MRR
AmaLgam [15]	2014	*BugLocator dataset [9]*	hand-tuned ensemble	Top N, MAP, MRR
Ye et al. [3]	2014	AspectJ, BIRT, Eclipse, JDT, SWT, Tomcat	LtR: SVMrank	Top N, MAP, MRR
HyLoc [19]	2015	*Ye et al. dataset [3]*	rVSM + DNN (RBM)	Top N, MAP, MRR
AmaLgam+ [16]	2016	*BugLocator dataset [9]*	genetic algorithm: JGAP	Top N, MAP, MRR
Ye et al.+ [12]	2016	*Ye et al. dataset [3]*	LtR: SVMrank	Top N, MAP, MRR
ConCodeSe [20]	2016	AspectJ, ArgoUML, Eclipse, SWT, Tomcat, ZXing	VSM + heuristics	Top N, MAP, MRR
NP-CNN [21]	2016	AspectJ, Eclipse, JDT, PDE	convolutional DNN	Top N, MAP, AUC
BLIA [17]	2017	Aspectj, SWT, ZXing (*from BugLocator [9]*)	hand-tuned, 2-correlation	Top N, MAP, MRR
Shi et al. [13]	2018	Eclipse, SWT, ZXing *(from BugLocator [9])*	LtR: RankLib all	Top N, MAP, MRR

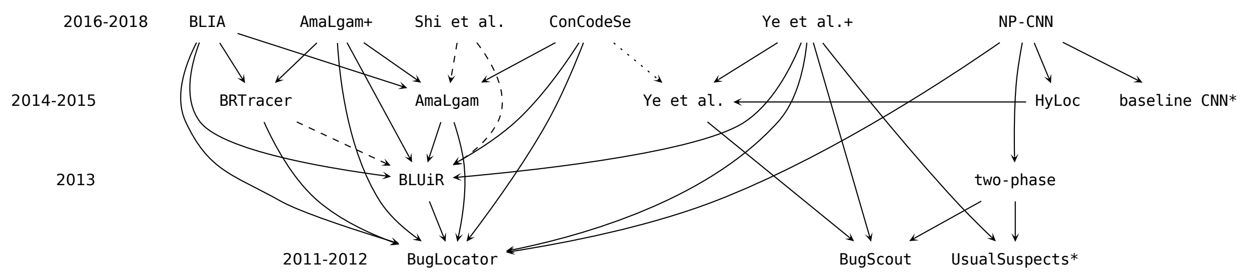


Fig. 1. A graph representation of improvements in the related work. Solid arrows point from an improving work to the improved one, dashed arrows represent a partial improvement or similar results, while dotted arrows show an improvement but comparison for only a few projects. Some papers compare to naïve baseline algorithms that are marked with * on the plot. BRTracer [11] improves on BLUiR [10] except SWT project. Shi et al. [13] replicated the AmaLgam [15] and BLUiR [10], but were unable to obtain results reported with original papers. ConCodeSe [20] compared only Tomcat project with Ye et al. [3].

Algorithms based on artificial deep neural networks (DNN) aim to bridge the lexical gap between bug reports and source code. The key limitations is the scalability, as DNN-based approaches often require a huge number of examples and features [19]. Furthermore, they often consume significantly more time and memory than previously listed methods including our proposed algorithm. As a consequence, processing of a single bug report takes minutes [21] on modern hardware.

HyLoc [19] is based on combining deep neural network with textual similarity model. It uses neural network to find pairs of related terms between reports and source code files.

Huo et al. [21] propose NP-CNN algorithm based on Convolutional Neural Network, with the goal of utilizing the program structure and natural language processing simultaneously. Special cost function was defined to account for the imbalance between the number of defect and non-defect files.

Two-phase algorithms include introductory classification of reports. In this setup, first phase determines which classifier (if any, as some bug reports may be dropped) will be used for the final prediction. The main idea is to create ranking algorithms that specialize in certain types of bug reports and a classifier that determines which algorithm will be used for the given bug report. This may improve results but increases the complexity of the ranking model. Furthermore, it does not solve the problem of the changing characteristics of the project as in our proposed algorithm.

The two-phase algorithm proposed by Kim [18] aims to find files relevant to a specific bug report by training Naïve Bayes on a bag-of-word features obtained from previous bugs. The authors use the probability per each file to select top k files related to bug report. To enhance the performance separate grading classifier was used to filter out bug reports that do not contain enough information to predict files.

An example of an adaptive approach is QUEST [23] proposed by Moreno et al. which recommends best IR algorithm for a given bug report, based on report characteristics. It was evaluated on small dataset of 12 software projects maintained by the Apache Software Foundation, using Top N and rank of the first relevant result. On average QUEST improves or preserves quality for 76% of the queries compared to other IR algorithms used standalone.

3 The Algorithm

Given a bug report, our algorithm performs the bug localization by computing the likelihood score for each file in the repository at the time of the bug report creation. This score is used to create ranking list of files that are most likely the cause of the bug. More probable files will get a higher score.

The algorithm does not require additional steps to fine tune parameters to the project in a separate parameter fitting

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

stage [9], [10], [12]. We only assume that there is some history of a project development and bug reporting. Moreover, as projects mature, and more parties are involved in the development process, the characteristic of the project will change. We consider it as an important aspect, thus, we designed the model so it could adapt to the project over time. The adaptation process will follow the development process of the project, rather then adapting to the characteristics of a bug report, like in Moreno et al. paper [23].

3.1 Features

Bug localization models operate on a set of features that captures relationships between files and bug reports. A large number of features was proposed over time (see e.g. Ye et al.+ [12] and references therein). Each pair of a bug report and a source file (r, s) is represented as a vector of k features: $\mathbf{\Phi}(r,s) = [\phi_i(r,s)]_{1 \leq i \leq k}$. We use the same set of 19 features as Ye et al.+ [12], following their naming and numbering, as listed in Table 2, except for ϕ_2, which we modified. We slightly adjusted formal definitions of these features to make it simpler and more accurately reflect their meaning. The features are normalized using standard min–max scaler $\frac{\phi_i - \min_n \phi_i}{\max_n \phi_i - \min_n \phi_i}$, with $\phi_i(r,s)$ values from current fold n that are below or above values from the previous fold, that is $\min_n \phi_i$ and $\max_n \phi_i$, set to 0 or 1, respectively [3].

We reimplement Ye et al.+ [12] feature extraction procedure using Python and Java (the latter for `ASTParser`). To build the models we used Pandas [24], Sklearn [25], Numpy [26], NetworkX [27] and Natural Language Toolkit (NLTK) [28]. The file dependency graph was created by parsing all Java files using the Eclipse ASTParser.

Main differences compared to Ye et al.+ [12] are new ϕ_2^* feature and different TF-IDF weight scheme. Read Section 4 to see how this impacts the results. We publish the feature preparation code and the values of resulting features as a supplement to the Ye et al. dataset [3], [12].

3.1.1 A rationale for the new feature ϕ_2^*

Feature ϕ_2 proposed by Ye et al. [3], [12] (see Table 2) is constructed based on API description extracted from the project documentation. While this feature can enhance overall process of bug localization, the way it is constructed in the original papers [3], [12] poses two threats. First, it may leak information as it is based on snapshots of documentation, thus some bug reports may be evaluated against the documentation not available at the time of the report. For example the earliest bug found in `AdaptableList.java` (bug id 5964) was reported in 2001 but the API documentation used by Ye et al. [3], [12] describes features added to the project in 2004 (ver. 3.0) and 2011 (ver. 3.1). Second, it contains only a subset of entries available in the original documentation and no justification is given for why some documents are excluded. For example, an important class for plugin development `UIPlugin` from Eclipse Platform UI is not included in the snapshot.

We propose a new feature ϕ_2^* that mitigates mentioned threats by using the documentation for API derived from AST of the source code available at the time when the bug report was submitted, $s.api^*$. More computational effort is required compared to original ϕ_2, but it does not leak information from the future, and is not subjective to the initial API selection.

3.1.2 Text processing differences

Small differences exists between the Ye et al. papers [3], [12] and the feature extraction code we received from its authors. Their implementation uses WordNet lemmatizer, and manually adjusted stop words per each project, which was not described in the original paper [3]. We follow the procedure from Ye et al. paper [12].

We decided to use an established TF-IDF weighting scheme used in the scikit-learn library [25] and in the Apache Lucene search engine. Given term t, document d and set of all documents D, $\textit{tf-idf}(t,d,D) = \textit{tf}(t,d) \cdot \textit{idf}(t,D)$, where $\textit{tf}(t,d)$ is raw count of term t in document d and $\textit{idf}(t,D) = \log \frac{1+|D|}{1+|\{d : t \in d\}|} + 1$.

3.2 Metrics

Training and testing of models relies on the quality metrics of ranking. There are several quality measures (*metrics*) which are commonly used to judge how well a ranking algorithm performs, we follow definitions from Ye et. al. [12]. Here, let Q denote the set of all queries (bug reports), $q \in Q$ be a single query, and K be the set of all relevant files (fixed files).

Accuracy@k, also known as likelihood, is based on Top K, and measures the percentage of bug reports for which the model predicted at least one correct recommendation in the top k ranked files and is defined as:

$$\text{Top } K = \#\text{ at least one correct in top } k, \tag{1}$$

$$\textit{Accuracy@k} = \frac{\text{Top } K}{|Q|}. \tag{2}$$

Mean Average Precision (MAP) metric is defined as

$$MAP = \sum_{q \in Q} \frac{AvgP(q)}{|Q|}, \tag{3}$$

where the $AvgP = \sum_{k \in K} \frac{Prec@k}{|K|}$ is the average precision and $Prec@k = \frac{\#\text{correct } q \text{ in top } k}{k}$.

Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of the first correct answer for a sample of queries:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{rank}(\textit{first}(q))}, \tag{4}$$

where $\text{rank}(\textit{first}(q))$ is the position of the first relevant document in the ranked list for query q.

3.3 Learning to rank: the setup

Using learning to rank approach in the context of bug localization requires a special setup. The main challenge is the huge imbalance between relevant and irrelevant files for a given bug report. For example, 4 largest projects from Ye et al. dataset [12] have between 4000 and 6000 files, while median of relevant files for a bug report is between 1 and 3. Thus, a special data preparation is required in order to mitigate the imbalance problem. Based on related works we generalized the typical setup used in this case (see Fig. 2): the *initial ranking*, the *training target* and the *imbalance handling*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

TABLE 2
Features used in ranking model - proposed by Ye et al.+ [12]. **Notation:** sim is the cosine similarity. *Query dependent* features relay on both the source code s and on the bug report r. We use new notation for features $\boldsymbol{\phi_2}$, $\boldsymbol{\phi_3}$, $\boldsymbol{\phi_5}$. We propose ϕ_2^* instead of ϕ_2, see Section 3.1.1 for rationale.

Feature	Description	Formula	Query dep?
ϕ_1	Surface lexical similarity	$\phi_1(r,s) = \max\big(\{sim(r,s)\} \cup \{\, sim(r,m) \mid m \in s\,\}\big)$	Yes
$\boldsymbol{\phi_2}$	API-enriched lexical similarity	$\phi_2(r,s) = \max\big(\{sim(r,s.api)\} \cup \{\, sim(r,m.api) \mid m \in s\,\}\big)$	Yes
$\boldsymbol{\phi_3}$	Collaborative filtering score	$\phi_3(r,s) = sim\big(r, \text{concat}(\{\, r.summary \mid r \in br(r,s)\,\})\big)$	Yes
ϕ_4	Class name similarity	$\phi_4(r,s) = \lvert s.main\_class\rvert \cdot \mathbb{1}[s.main\_class \in s.summary]$	Yes
$\boldsymbol{\phi_5}$	Bug-fixing recency	$\phi_5(r,s) = \big((r.date - last(r,s).date).months + 1\big)^{-1}$	Yes (Timestamp)
ϕ_6	Bug-fixing frequency	$\phi_6(r,s) = \lvert br(r,s)\rvert$	Yes (Timestamp)
ϕ_7	Summary–class names similarity	$\phi_7(r,s) = sim(r.summary, s.class)$	Yes
ϕ_8	Summary–method names similarity	$\phi_8(r,s) = sim(r.summary, s.method)$	Yes
ϕ_9	Summary–variable names similarity	$\phi_9(r,s) = sim(r.summary, s.variable)$	Yes
ϕ_{10}	Summary–comments similarity	$\phi_{10}(r,s) = sim(r.summary, s.comment)$	Yes
ϕ_{11}	Description–class names similarity	$\phi_{11}(r,s) = sim(r.description, s.class)$	Yes
ϕ_{12}	Description–method names similarity	$\phi_{12}(r,s) = sim(r.description, s.method)$	Yes
ϕ_{13}	Description–variable names similarity	$\phi_{13}(r,s) = sim(r.description, s.variable)$	Yes
ϕ_{14}	Description–comments similarity	$\phi_{14}(r,s) = sim(r.description, s.comment)$	Yes
ϕ_{15}	In-links = # of file dependencies of s	$\phi_{15}(r,s) = s.inLinks$	No
ϕ_{16}	Out-links = # of files that depend on s	$\phi_{16}(r,s) = s.outLinks$	No
ϕ_{17}	PageRank score	$\phi_{17}(r,s) = PageRank(s)$	No
ϕ_{18}	Authority score (HITS)	$\phi_{18}(r,s) = Authority(s)$	No
ϕ_{19}	Hub score (HITS)	$\phi_{19}(r,s) = Hub(s)$	No
$\boldsymbol{\phi_2^*}$	**full API–enriched lexical similarity**	$\phi_2^*(r,s) = \max\big(\{sim(r,s.api^*)\} \cup \{\, sim(r,m.api^*) \mid m \in s\,\}\big)$	Yes

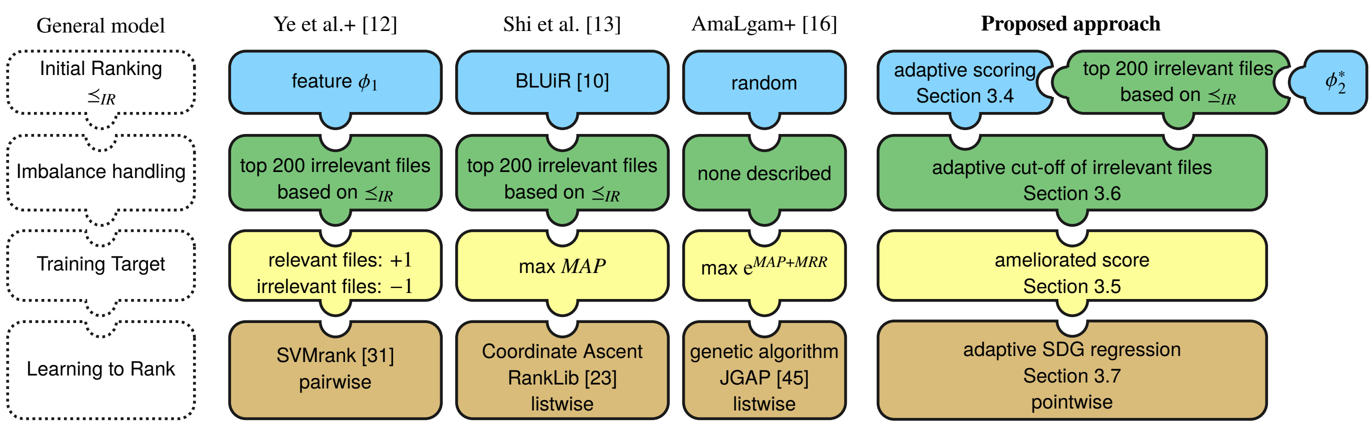


Fig. 2. Learning to rank in bug localization: generalized block schema (dotted lines), comparison of existing approaches (Ye et al.+ [12], Shi et al. [13], AmaLgam+ [16]) and the proposed algorithm.

The *initial ranking* selection is a preliminary step used either in the *imbalance handling* algorithm or for establishing the *training target*. Examples of *initial rankings* used are: the feature ϕ_1 in Ye et al.+ [12], results of BLUiR [10] algorithm used in Shi et al. [13], and a random order for the genetic algorithm in AmaLgam+ [16]. We propose a custom approach, described in Section 3.4, where we use a score based on (per–fold) adaptive feature weights approach. Then, there is the *imbalance handling* algorithm which cuts off most of the irrelevant files making the training set more balanced. In both Ye et al.+ [12] and Shi et al. [13] all but top 200 of irrelevant files based on the *initial ranking* are excluded from the training set. In AmaLgam+ [16] there is no imbalance handling. We use a two step approach, described in Section 3.6, where we first cut using the feature ϕ_2 as the ranking and use top 200 irrelevant files. Then we proceed with an adaptive cut-off which uses the *initial ranking* to further reduce the number of files. Finally, there is the *training target*, that will be used in the training process. The authors of AmaLgam+ [16] maximize the function $e^{MAP+MRR}$ on results of a genetic algorithm selecting weights on randomly selected 5% of the dataset. Shi et al. maximize the MAP metric [13]. In Ye et al.+ approach [12] a simple training target is used, based on a binary relation i.e. $+1$ for relevant files and -1 for irrelevant files. On the other hand, we use more sophisticated mechanism based on an ameliorated score, described in Section 3.5. This leads to a fine-grained ranking that induces a linear order.

3.4 Initial ranking: the adaptive scoring algorithm

Pointwise learn–to–rank approach (see Section 2) needs a good target score or a target order; this will be explored in more detail in Section 4.3.4. We will base such training target (described in Section 3.5) on a set of different heuristics based on building the relevancy score for file-bugreport pairs coming from an importance analysis of features.

This algorithm is intended as a preliminary step for pointwise learn–to–rank approach, but due to its efficiency

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

it may be treated as a separate standalone algorithm, see Section 4.3.1.

3.4.1 The scoring functions

To create the target score we propose several scoring functions consisting of various statistical tests, classification methods or information theory measures. All scoring functions are a linear combination of features $p_i(r,s) = \sum_k w_k \cdot \phi_k(r,s)$, where weights w_i are set heuristically, based on several proposed criteria. Those functions are then evaluated on each training sample for the fold, which is the same as the training subset for the fold in Ye et al.+ [12].

Heuristics for weights are based on the estimation of how well each feature ϕ_i can distinguish between relevant and irrelevant files per bug report. The intuition is that weights can be used to rank importance of each feature in a similar fashion to how they are used for feature selection [30]. We summarize all used functions and weights in Table 3. Weights are scaled by $1/\sum w_i$ for given scoring criteria.

First group of heuristics is based on statistical tests. We split each feature into relevant and irrelevant groups and then use a statistic as a measure of corresponding feature importance. That said we use Kruskal-Wallis H-test [31] to test whether the population medians are equal, the T-test for independent samples [32] to check if means are significantly different, chi-square test [32] to find out features independent of relevancy class, and Levene test for equality of variances [33] to assess the equality of variances between relevant and irrelevant groups.

In the next group we have various classification algorithms used to assess feature importance. Each classifier is trained to predict fix and non-fix files. We use AdaBoost tree classifier [34], [35], Extremely Randomized Trees classifier [36] and Gradient Boosting regression [37].

Finally, we have heuristics based on the index of dispersion [32], maximum absolute deviation [32], and mutual information [38]. The use of index of dispersions is motivated by the idea that better distinguishing features have different distribution for relevant and irrelevant files. We also check the maximum absolute deviation variances calculated on the features of relevant files to capture the characteristics specific only to those files. Mutual information is used to check for the most discriminating features in the context of target variable.

The simplest heuristic is based on constant weights, i.e. the score is the sum of features ($w_i = 0.5$).

3.4.2 Adaptation: selection of the best model

We prepare several sets of normalized feature weights w_i, and choose the best scoring function (see Fig 2) according to *MAP* metric, as defined in Section 3.2. In order to select the best model, we utilize the 2-way cross validation during training evaluation for scoring. Each training fold of 500 bug reports is split into two equally sized disjoint parts. For each ordering of parts, we use one part to find the weights, and the other part to obtain metrics. This way we obtain metrics on all training bug reports, and use it to select the best scoring. We combine the best scoring weights from two datasets using the mean to create one set of weights for the whole training fold. The process is repeated for each training fold, allowing for various models to be selected. We decided to use adaptation because training folds tend to change their characteristics (see Section 4.3.6 for examples).

3.5 Training target: ameliorated score

Bug localization that use coarse–grained binary relation in the pointwise learning-to-rank setup performs poorly as reported by Shi et.al. [13]. To remedy this, we add the maximum of feature values to the score of relevant files to construct a fine-grained ameliorated score function:

$$p^*(r,s) = \sum_{i=1}^{n} w_i \cdot \phi_i(r,s) + \mathbb{1}_{[s \in \mathit{fixes}(r)]} \cdot \max_{i=1..n} \phi_i(r,s),$$

where $\mathit{fixes}(r)$ is a set of fixed files for a given bug report r. This modification ensures proper training order, by ensuring that all fixed files have greater p^* score than non–fixes. Both p and p^* allow fine-grained comparison between files for given bug report.

3.6 Imbalance handling: adaptive cut–off

Files not related to a bug report severely outnumber buggy files. To be able to correctly train regression models we need to create a training set with an appropriate ratio of bug related and unrelated files (i.e. all files with bugs, and small sample of other files). Ye et al.+ [12] proposed to take only 200 irrelevant files based on the ϕ_1 feature, based on experiments with two projects. Similar approach with 200 threshold was also utilized by Shi et al. [13]. While using the threshold of 200 lowers the number of irrelevant files, there is still an imbalance between relevant and irrelevant classes. We want the sizes of created classes to be of similar magnitude. First we narrow the files by taking the top 200 irrelevant files based on the ϕ_2^* feature. In case of less than 200 files we take all irrelevant files present. We selected the ϕ_2^*, as we were able to obtain better results with this feature, than with ϕ_1 or ϕ_2. Then we further narrow the training set with the cut-off function,

$$t_f(r) = \mathit{fixes}(r) \cup \{s \notin \mathit{fixes}(r) \mid 0 < p^*(r,s) \leq c_f(r)\},$$

which selects non-defect (irrelevant) files for the training set. It takes source files with the lowest score per bug report, up to a given fraction of $f\%$ of the irrelevant files in training set (used for 3.4). We evaluate each variation of regression models with a cutoff function using 5%, 10%, 15%, 20%, 25% and 30% of previously chosen irrelevant files per bug report. See Section 4.3.6 and Fig. 6 for results for cutoff adaptation.

3.7 Pointwise learning to rank algorithm

Having the fine grained ameliorated scoring for the training dataset enables us to efficiently utilize pointwise learning–to–rank approaches based on regression.

Similarly to the adaptive scoring step (Section 3.4), we test several regression models. Specifically we use several variants of Stochastic Gradient Descent regression [39], with following loss functions: ordinary least squares [39], Huber [40], epsilon insensitive [41] and squared epsilon insensitive [41], and with following regularization terms: no regularization, L1, L2, or Elastic Net.

The regression model is then trained and evaluated using 2-way cross validation. The models search space consists

TABLE 3
Scoring functions and weight schemata used in the scoring phase, where ϕ_i - file values for feature i, ϕ_i^{fix} - values for feature i for files used in fix, ϕ_i^{irr} - values for feature i for files not used in fix (irrelevant), Y - true/false values for each file, true if file is a fix, else false

Based on	Weights
W statistics from Levene test for equality of variances with median as center function [33]	$w_i = W_{\phi_i} / \sum_{j=1}^{n} W_{\phi_j}$
H statistics from Kruskal-Wallis H-test [31]	$w_i = H_{\phi_i} / \sum_{j=1}^{n} H_{\phi_j}$
T statistics from T-test for independent samples [32]	$w_i = T_{\phi_i} / \sum_{j=1}^{n} T_{\phi_j}$
χ^2 statistics from chi-square test [32]	$w_i = \chi_{\phi_i} / \sum_{j=1}^{n} \chi_{\phi_j}$
Features weights computed by AdaBoost SAMME.R Classifier [34], [35]	model specific
Features weights computed by Extremely randomized trees classifier [36]	model specific
Features weights computed by Gradient Boosting regression [37]	model specific
Mutual Information between Discrete and Continuous Data Sets [38] denoted as I	$w_i = I(\phi_i, Y) / \sum_{j=1}^{n} I(\phi_j, Y)$
Index of dispersion [32]	$w_i = {}^{D_{\phi_i^{fix}}}/{}_{D_{\phi_i^{irr}}}$ where $D_{\phi_i} = {}^{\sigma^2_{\phi_i}}/{}_{\mu_{\phi_i}}$
Maximum absolute deviation variances [32]	$w_i = \text{mean}\,(\lvert\phi_i^{fix} - \max \phi_i^{fix}\rvert)$
Predefined set of weights	$w_i = 0.5$

of Cartesian product of all possible variants of training models and cut-off thresholds. The winning model is selected based on resulting *MAP* metric. The selected parameters are presented in Section 4.3.6.

4 Evaluation

We pose and answer four research questions that evaluate the effectiveness of our approach compared to other fault localization techniques.

RQ1: Can we outperform other bug localization techniques?
RQ2: Can pointwise learning to rank be competitive with pairwise and listwise approaches used for bug localization?
RQ3: Is the proposed algorithm sufficiently efficient for the intended use?
RQ4: What is the impact of our changes to the procedure given by Ye et al.+ [12]?

4.1 Evaluation Design

For the evaluation we use two commonly used datasets: the fine-grained [12] and the BugLocator [9]. We examine 7 projects: all 6 projects from the fine-grained dataset [12], and Eclipse project from the BugLocator [9] dataset.

TABLE 4
Publicly available benchmark datasets used in this article.

Dataset	Project	# of bugs	appr. # files	Missing desc.
Ye et al.	Eclipse UI	6,495	3,454	19%
	JDT	6,274	8,184	15%
	BIRT	4,178	6,841	28%
	SWT	4,151	2,056	21%
	Tomcat	1,056	1,552	50%
	AspectJ	593	4,439	21%
BugLocator	Eclipse 3.1	3,075	12,863	
	SWT 3.1	98	484	
	AspectJ	286	6,485	
	ZXing	20	391	

To answer both **RQ1** and **RQ2** research questions we apply our algorithm to both datasets, then we compute metrics Accuracy@k, MAP and MRR to characterize the effectiveness.

For the fine grained dataset we utilize the same data split into training and testing subsets as Ye et al.+ [12], using equally sized folds of 500 bug reports, where consecutive folds are used for training and evaluation. This gives us two folds for AspectJ and Tomcat, eight folds for BIRT and SWT and twelve folds for Eclipse UI and JDT.

For completeness we include results reported by other projects on BugLocator dataset i.a., BugLocator [9], BRT-Tracer [11], BLUiR [10], Ye et al.+ [12], AmaLgam+ [16], ConCodeSe [20] and Shi et al. [13][2]. It should be noted, that Shi et al. [13] replicated results of BLUiR [10] and AmaLgam+ [16], with lower results than originally reported, we decided to cite the original findings. Similarly to Ye et al.+ [12], we use the Eclipse 3.1 project (as the other projects are not big enough for training), and split the data into six consecutive folds with 500 bug reports in each. We use fold n to train fold $n+1$, except for the first fold, for which the second fold is used for training.

In order to answer **RQ3** we measured training time for our algorithms, i.e. all steps described in Section 3, including parameter adaptation using the fine grained dataset.

To answer **RQ4** we evaluated the replication code for Ye et al.+ algorithm [12] with different variants of features using the fine grained dataset.

4.1.1 *Fine-grained dataset*

The fine-grained dataset was proposed by Ye et al. [3], [12], and is publicly accessible[3]. It closely resembles real life use cases; thus it should be preferred over commonly used BugLocator dataset [9] (see discussion in Ye et al.+ [12]). It consists of six open-source Java projects: Eclipse Platform UI, JDT (Java Development Toolkit), BIRT (Business Intelligence and Reporting Tools), SWT (Standard Widget Toolkit), Tomcat and AspectJ. All these use Bugzilla as the issue tracking system and Git as the version control system. Only bug

2. For papers that reported Top-N the conversion was made to Accuracy@k. For Ye et al.+ [12] we only report results from their extended work as they improve initial results.
3. http://dx.doi.org/10.6084/m9.figshare.951967

reports with clear corresponding fixed files were considered, i.e. bug reports are paired with bugfix commits by searching commit messages for special phrases such as "bug 31946" or "fix for 31946" according to the heuristics proposed by Dallmeier et al. [42]. As in Ye et al. papers [3], [12], the first revision before the fix was used as the substitute of the exact version for which the bug was reported.

While investigating the replication we discovered that for some bug reports the description is missing in the dataset, but it is present in the Bugzilla. The percentages of missing descriptions per project are presented in Table 4. We informed Ye et al. [3], [12] about this defect, unfortunately they are uncertain at which point it was introduced, during the construction of the dataset or the export of its public version. Therefore, it remains uncertain if missing descriptions were used in Ye et al. [3], [12] (see Section 4.3 for further analysis). To overcome this problem we decided to evaluate our algorithm on datasets both with and without missing descriptions.

4.1.2 BugLocator dataset

The BugLocator dataset [9] is commonly used by variety of the state–of–the-art algorithms, and can be found as part of the Bug Center project[4]. This dataset is based on a single version of project sources connected to multiple fixed bug reports, and contains fewer bug reports than the fine-grained dataset [12]. The values of features are not present as well. It does not contain any repository history or any explicit bug fix connection using commit SHA-1 identifier. Only the date of the fix commit is available. Additionally, bug reports for Eclipse 3.1 project are related to multiple existing repositories such as Eclipse Platform UI or JDT. This leads to misaligned file paths between bug reports and repositories, and might affect algorithm performance[5]. Ye et al.+ [12] also found files included in bug reports, but not present in the version of source code in dataset, as those files were deleted at some point in history.

4.2 Results

4.2.1 Effectiveness compared to others (***RQ1*** and ***RQ2***)

In Table 5, Table 6 and on Fig. 3 we compare results to learning-to-rank approach by Ye et al.+ [12]. We obtain better results in terms of Accuracy@1, MAP and MRR. Results for the rest of Accuracy@k are comparable (i.e. slightly worse or better, see Table 6 and on Fig. 3). We significantly improve results on AspectJ and BIRT projects. Adding missing descriptions yields better results in terms of Accuracy@k and MAP for AspectJ, Eclipse UI and Tomcat, while for other projects results are lower by about 1% on Accuracy@k. We outperform baseline algorithms BugLocator, VSM and UsualSuspect as reported by Ye et al.+ [12], both with and without missing description. Thus, our pointwise learning to rank algorithm gives comparable or better results than other bug localization techniques.

For BugLocator dataset we obtained compelling results, and outperform other approaches, with the exception of Accuracy@10 for Shi et al. [13]. In Table 7 we compare to other approaches, with learning to rank based approaches underlined. The results were taken directly from the corresponding publication, with the highest results highlighted. Due to the fact that this dataset contains a single version, we did not use the past history of the repository when constructing ϕ_2^*. Some experiments differ in setup, thus are not directly comparable [9] [11], [13] (see Table 7). We include them for completeness.

4.2.2 Performance for the fine-grained dataset (***RQ3***)

Training times are below the main competitor algorithm [12], but were made using more recent hardware. Execution times are gathered in Table 8. No other load was present during experiments. The computation time of features was measured on *Dataset Setup* (similar setup to the one used by Ye et al.+ [12]) and is comparable to Ye et al.+ [12] in terms of the average time per bug report. The computation of features is the most time consuming step, but the average time is acceptable for the intended use.

4.2.3 Differences in the construction of features (***RQ4***)

We investigated the impact of changes in TF-IDF weighting schemes and tokenization (see Section 3.1.2), and replacement of ϕ_2 with ϕ_2^* (see Section 3.1.1) on the replication of Ye et al.+ [12]. For example for Tomcat project this improves MAP by 3% and MRR by 2%, but for SWT project this lowers MAP and MRR by 1%. We conclude that observed improvements for proposed algorithm are higher than possible improvements gained from new variants of features. Additionally, in some cases new features variants could lower the score, as original feature ϕ_2 from Ye et al.+ [12] was carrying information back from the future to the past. Proposed feature ϕ_2^* fixes the leak which may lead to lower results as in the case of SWT.

4.3 Discussion

4.3.1 Adaptive scoring (***RQ1***)

A part of our algorithm, the adaptive scoring (see Section 3.4) can be used as a standalone ranking algorithm. It outperforms Ye et al.+ [12] on AspectJ, Birt, JDT and SWT, with MAP of 0.45, 0.21, 0.40 and 0.41 respectively on Ye et al.+ [12] dataset with missing descriptions. For Eclipse Platform UI and Tomcat it obtains MAP of 0.43 and 0.48, slightly below Ye et al.+ [12]. Similarly, it is able to outperform other state–of–the–art approaches on BugLocator dataset [9], with MAP of 0.58 on Eclipse 3.1, slightly below our main algorithm.

4.3.2 Imbalance handling (***RQ1***)

Selection of which non-defect (irrelevant) files to include in the training set affects learning performance. For the training set Ye et al.+ [12] select irrelevant files *similar* to the bug report, with the highest value of the cosine similarity of the file content and the text of the bug report (feature ϕ_1). They have found out that using up to around 200 irrelevant files per bug report improved the MAP measure. Shi et al. [13] selected irrelevant features for training utilizing BLUiR [10] algorithm. The weights used in BLUiR [10] are trained on the whole AspectJ from the BugLocator dataset [9]. The ConCodeSe [20] uses *randomly* selected 2.2% of bug reports as training set to adjust report description similarity weights

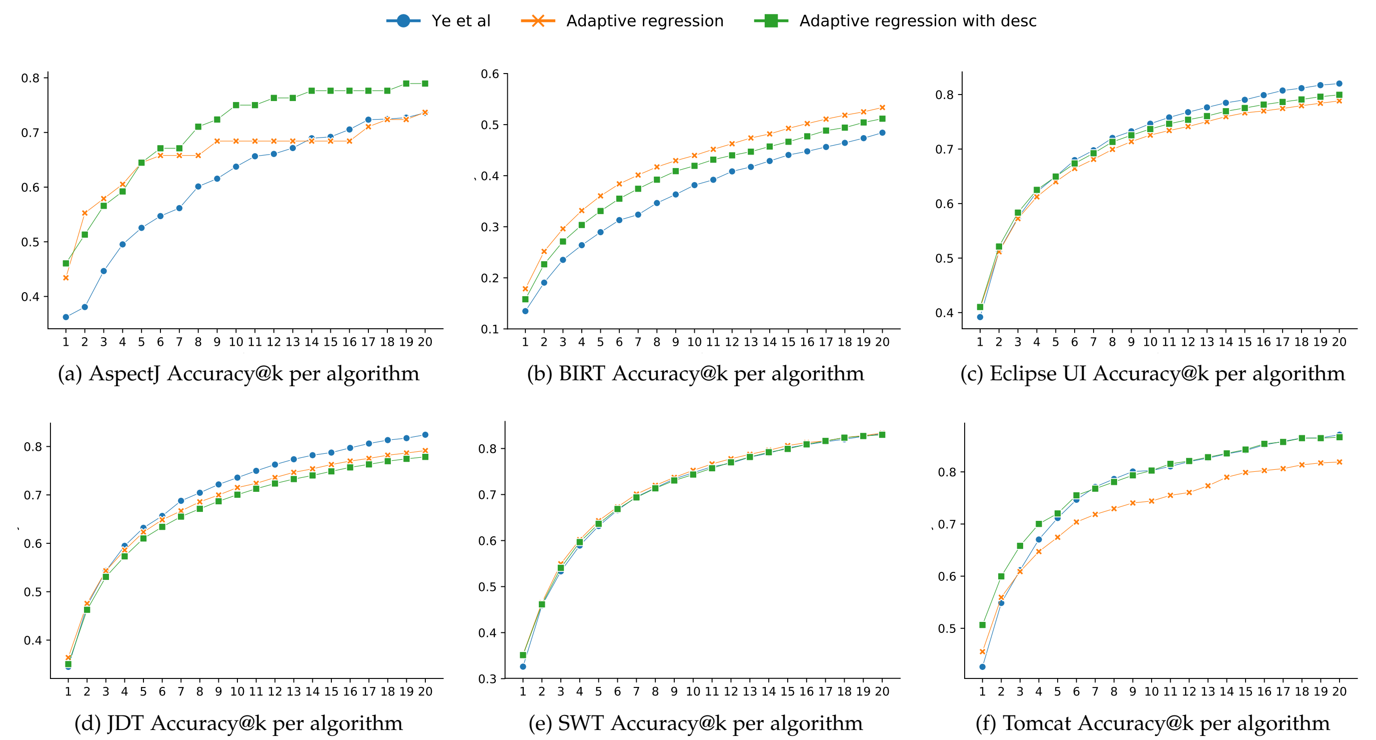
4. https://code.google.com/archive/p/bugcenter/

5. For instance "`CCombo.java`" class from Eclipse SWT project, which has package name instead file path present in the dataset.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

TABLE 5
MAP and MRR evaluation on fine-grained Ye et al. dataset [12], including results reported therein. The best and second best results are highlighted in grey and light grey respectively. Our algorithm is able to outperform Ye et al. on MAP and MRR on all projects. We do not present BugLocator, VSM and Usual Suspects, as Ye et al. outperform these algorithms. We denote the version with restored missing bug report descriptions as "with desc".

	MAP						MRR					
Algorithm	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat
Adaptive regression	0.45	0.21	0.44	0.40	0.42	0.50	0.53	0.27	0.52	0.48	0.48	0.56
Adaptive regression with desc	0.46	0.19	0.45	0.39	0.41	0.54	0.54	0.25	0.52	0.47	0.48	0.61
Ye et al.+ [12]	0.37	0.16	0.44	0.39	0.40	0.49	0.44	0.21	0.51	0.47	0.46	0.55



(a) AspectJ Accuracy@k per algorithm
(b) BIRT Accuracy@k per algorithm
(c) Eclipse UI Accuracy@k per algorithm
(d) JDT Accuracy@k per algorithm
(e) SWT Accuracy@k per algorithm
(f) Tomcat Accuracy@k per algorithm

Fig. 3. Accuracy@k (see Equation (1)) evaluation on fine-grained Ye et al. dataset [12] including results as reported in source publication [12]. Results for Ye et al. were acquired by digitalizing Accuracy@k diagrams from original paper [12] (as authors do not provide exact results). We omitted other commonly evaluated algorithms, BugLocator, VSM and UsualSuspect as they are outperformed by presented algorithms (see [12]).

(denoted TT score) and 9.4% of bug reports to find the report summary weights (denoted KP score). Authors of the AmaLgam+ [16] algorithm also randomly sample the whole dataset to find 5% bug reports for training.

We train, evaluate and select adaptive scoring algorithm (Section 3.4) on the training set including upper limit of 200 irrelevant files with highest values of ϕ_2^* feature (i.e. *similar* to the bug report). During the learn–to–rank step (Section 3.7) we further reduce our training set by selecting only a portion of *dissimilar* irrelevant files (with lowest ameliorated score), using adaptive cutoff threshold to maximize training results.

4.3.3 Fold size (**RQ1** and **RQ2**)

The size of the training fold is set the same way as in Ye et. al. [12] (500 bug reports) for comparison reasons. We also investigated how different fold sizes can affect the results. We used fold sizes of 100 and 250 bug reports on Birt, Eclipse UI, JDT and SWT projects from Ye et al. dataset [12], and achieved comparable results as with the default size.

4.3.4 Learn to rank in Bug Localization (**RQ2**)

The usefulness of pointwise, pairwise and listwise approaches in bug localization depends heavily on the training target function. Ye et al.+ [12] formulated the training target as relevant and irrelevant files. Such formulation is sufficient for pairwise and listwise approaches but does not fit well the pointwise approach. Ye et al.+ [12] uses SVMrank which is a pairwise learning to rank algorithm. Shi et al. [13] tested all suitable learning to rank algorithms from RankLib [22] (covering pointwise, pairwise and listwise cases), and concluded that the listwise based approach *Coordinate Ascent* which optimizes MAP metric is the best suited method. They also noted poor performance of algorithms based on the pointwise principle, we assume due to the formulation of the training target function. The AmaLgam+ [16] uses a genetic algorithm from the JGAP [43] library to optimize feature weights. It is a listwise learn to rank as $e^{MAP+MRR}$ is optimized.

In our algorithm we propose custom learning to rank

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

TABLE 6
Detailed Accuracy@1, Accuracy@5 and Accuracy@10 evaluation on fine-grained Ye et al. dataset [12]; results as reported in source publication [12]. Results for Ye et al. were acquired by digitalizing Accuracy@k diagrams from original paper [12] (as authors do not provide exact results). The best and second best results are highlighted in grey and light grey respectively.

Project	Algorithm	Acc@1	Acc@5	Acc@10
	Ye et al.+ [12]	0.362	0.526	0.637
Aspectj	Adaptive regression	0.434	0.645	0.684
	Adaptive reg. with desc	0.461	0.645	0.75
	Ye et al.+ [12]	0.135	0.289	0.382
Birt	Adaptive regression	0.179	0.36	0.44
	Adaptive reg. with desc	0.158	0.331	0.419
	Ye et al.+ [12]	0.392	0.65	0.746
Eclipse	Adaptive regression	0.409	0.64	0.726
	Adaptive reg. with desc	0.41	0.649	0.737
	Ye et al.+ [12]	0.345	0.632	0.736
JDT	Adaptive regression	0.364	0.623	0.715
	Adaptive reg. with desc	0.35	0.61	0.7
	Ye et al.+ [12]	0.326	0.632	0.747
SWT	Adaptive regression	0.352	0.643	0.753
	Adaptive reg. with desc	0.351	0.636	0.743
	Ye et al.+ [12]	0.426	0.712	0.803
Tomcat	Adaptive regression	0.455	0.675	0.744
	Adaptive reg. with desc	0.506	0.72	0.803

TABLE 7
Algorithm evaluation on Eclipse 3.1 project from single version BugLocator dataset [9], including results of other algorithms using same dataset as in corresponding source publications. The best and second best results are highlighted in grey and light grey respectively. Underlined are learning to rank algorithms. Setup differences: † training of randomly selected 5% reports, ‡ manually adjusted weights, ‡‡ usage of 30 folds, 100 bug reports each.

Algorithm	Acc@1	Acc@5	Acc@10	MAP	MRR
BugLocator [9]	0.291	0.538	0.626	0.3	0.41
BRTTracer [11]	0.326	0.559	0.652	0.33	0.43
BLUiR [10]	0.329	0.562	0.654	0.33	0.44
Ye et al.+ [12]	0.34	0.57	0.66	0.34	0.45
AmaLgam+ [16]†	0.357	0.603	0.691	0.36	0.47
ConCodeSe [20]‡	0.376	0.612	0.699	0.37	0.57
Shi et al. [13]‡‡	0.297	0.664	0.85	0.306	0.399
Adaptive regression	0.7	0.752	0.785	0.6	0.728

training target function. Our target function is based on a score that assigns a value from the continuous range $[0, 1]$, which we then ameliorate to ensure that relevant files are on top of the list. All the relevant files will therefore have higher scores than irrelevant files, but we can also distinguish between two relevant (or irrelevant) files. The reformulation of training function allowed us to successfully apply pointwise learning to rank approach.

4.3.5 *Adaptation process (**RQ2**)*

To select optimal parameters, Ye et al.+ [12] conducted grid search on one training fold to find parameters for evaluation. Shi et al. [13] used default parameters present in RankLib [22] for all tested algorithms, with an additional manual grid search conducted on the first fold for Eclipse 3.1. The authors searched for Coordinate Ascent parameters restarts and iterations, choosing 5 and 25 to be used for the

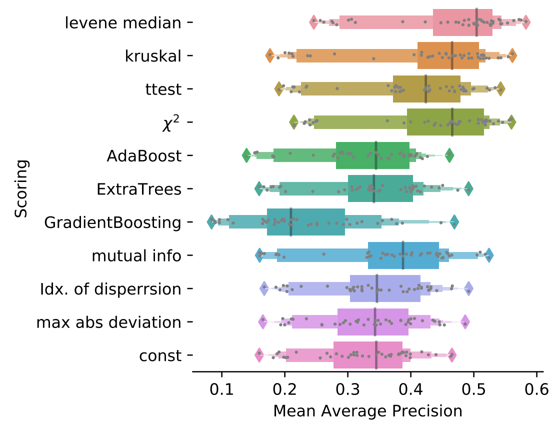


Fig. 4. MAP distribution on all training folds for all projects for different scoring functions. Distribution presented with letter-value plots [44], actual measurements shown as dots.

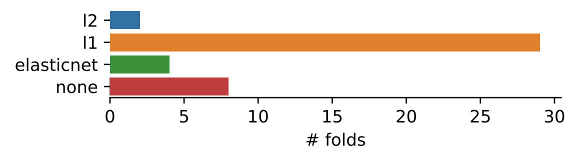


Fig. 5. Selected regularization terms for adaptive regression model across all training folds on fine-grained dataset [12].

rest of the project. The AmaLgam+ [16] computes weights using genetic algorithm optimizing $e^{MAP+MRR}$ on train data, using default parameters of JGAP library [43]. The characteristics of a project may change over time; thus, one model with predefined parameters may have suboptimal results for the whole dataset. Our algorithm is able to efficiently adapt the parameters over time from a predefined set of parameters. The parameters are selected on the training fold (fold n) and then used on the testing fold (fold $n + 1$).

4.3.6 *Selected parameters (**RQ2**)*

Depending on the dataset different parameters were selected by the adaptation process. The selection for the fine-grained dataset was: Levene test as the scoring function (see Fig. 4) and Huber as the loss function [40] with the cutoff factor of 5% (see Fig. 6). For two projects the folds changed characteristic. In case of Eclipse Platform UI the Kruskal-Wallis H-test [31] and T-test for independent samples [32] were used for two and one fold respectively. The chi-square test [32] was selected for one fold of JDT project. For the rest of folds the Levene test was selected. The dominant regularization function was the L1 norm as seen on Fig. 5. For the single version dataset the selection was: scoring – the Levene test (with one exception, where chi-square test was used); the loss function – Huber; cutoff – factor of 5%.

5 Threats to validity

In this section we explain the potential threats to validity of our research work.

Missing descriptions in datasets. All projects in dataset [12] were missing some bug report descriptions (see Section 4.1.1) that were present in Bugzilla. We have

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3064447, IEEE Transactions on Software Engineering

TABLE 8
Times are reported for fine-grained Ye et al. dataset [12]. Measurements are done using Python's `timeit` module. **Training:** Average and standard deviations of time in seconds per bug report and per file, using 10 runs on each project; done on *Evaluation Setup*. **Feature computation:** Summary feature computation time per bug reported; done on *Dataset Setup.Evaluation Setup:* 2 Ten-Core Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz processors, 62 GB RAM. *Dataset Setup:* 2 Six-Core AMD Opteron™ processors 2431 @ 2.40GHz, 32 GB RAM.

	Time	per	AspectJ	BIRT	Eclipse UI	JDT	SWT	Tomcat
Training	Regr.	bug rep.	0.321 (±0.003)	0.338 (±0.002)	0.345 (±0.003)	0.353 (±0.008)	0.328 (±0.002)	0.330 (±0.002)
		file	0.018 (±0.000)	0.017 (±0.000)	0.019 (±0.000)	0.018 (±0.004)	0.031 (±0.000)	0.020 (±0.000)
	Scor.	bug rep.	0.079 (±0.001)	0.080 (±0.000)	0.081 (±0.000)	0.082 (±0.001)	0.080 (±0.001)	0.080 (±0.001)
		file	0.004 (±0.000)	0.004 (±0.000)	0.004 (±0.000)	0.004 (±0.000)	0.008 (±0.000)	0.005 (±0.000)
Feat comp.		bug rep.	39.16	74.69	42.61	68.03	46.05	23.64

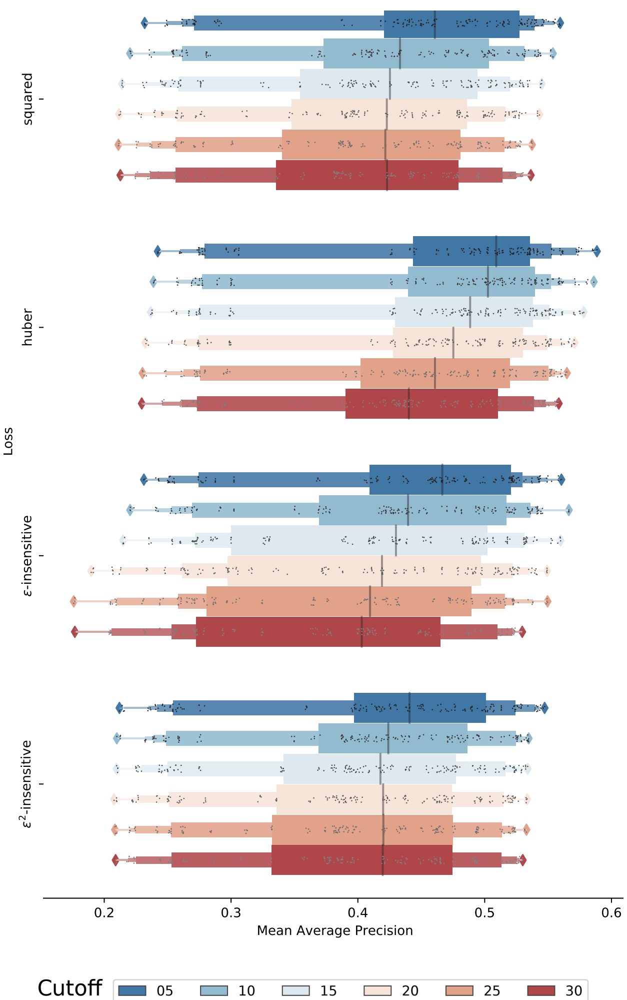


Fig. 6. Regression model MAP distribution on all training folds of fine-grained dataset [12] for each combination of loss function (Y-axis) and cutoff factor as % (cutoff's factors are grouped by loss, colours in groups are the in the same order as in legend). Distribution presented with letter-value plots [44], actual measurements shown as dots.

investigated the impact of missing descriptions by preparing a replication of Ye et al.+ [12] based on the source code the authors sent us. We evaluated results on all projects in fine-grained dataset [12]. The results are gathered in Table 9. Addition of descriptions improves Ye et al.+ replication results for 4 out of 6 projects, and decreases results for remaining 2 projects. For our algorithm it improves results for 3 projects and decreases results for 3. Note that for the SWT project the addition of missing descriptions lowers results of our algorithm, but improves results of Ye et al.+ [12] replication. For the rest of investigated projects the improvement or decrease is consistent between both algorithms. Our algorithm improves results by Ye et al.+ [12] regardless of whether the missing descriptions were in the dataset (see Table 5).

TABLE 9
Percentage impact of adding missing descriptions on MAP and MRR metrics for our Adaptive algorithm and Ye et al.+ [12] replication results.

Project	Adaptive MAP	Adaptive MRR	Replication MAP	Replication MRR
Aspectj	4%	2%	5%	6%
Birt	-11%	-9%	-8%	-8%
Eclipse UI	1%	1%	4%	4%
JDT	-3%	-3%	-3%	-3%
SWT	-1%	-1%	1%	2%
Tomcat	8%	8%	5%	5%

Homogeneity of projects in dataset. Most of the research in this topic (see Table 1) is done using two datasets [9], [12], which are based on open source Java projects. The majority of analysed projects are maintained by the Eclipse Foundation and use the same instance of Bugzilla bug tracker, and similar practises of code review. The question is how this impacts existing algorithms. There is a valid argument that a more diverse dataset is needed. It should include projects with different development practices and language specification.

Minimal project size. Our algorithm requires the presence of both bug reports and project commits in numbers sufficient for training. Projects with less than 100 bug reports and related commits (minimal tested fold size), such as micro-services, might not benefit from our bug localization algorithm. If multiple repositories of such micro-services are available in the shared industry setting, the problem can be alleviated by combining those repositories into one repository for training purposes, for instance by using "git submodule" utility. Similarly content of separate bug trackers can be joined to achieve the required number of bug reports.

Note that for the training set preparation we use the upper limit of 200 irrelevant files per bug report. In case of a project with less than 200 files, our algorithm will use all the present irrelevant files before applying the cutoff factor. Thus, for a project with at least two files the construction of a training set is possible.

Potential biases in dataset like bug reports which are wrongly classified or are already localized (i.e. filename or basename is present in the text of the bug report) should also be considered [45], [46]. We analysed the dataset from Ye et al.+ [12] for such biases. First, we found individual cases of bug reports pointing to wrong project due to an incorrect bug id in the commit message. Excluding them does not have impact on the overall results. Secondly, Kochar et. al. [45] found that in some projects for even 50% of bug reports may be already localized. This is not the case for Ye et al.+ [12], as around 70% of bug reports include at least one filename, but only 25% of them may be localized that way.

Retrospective vs prospective study. All presented papers in Section 2 focus on retrospective analysis of bug localization and our evaluation is done the same way. This allows comparable and replicable results. On the other hand, without prospective evaluation, involving new bug reports and real developers, it is hard to judge usefulness of proposed solution. Unfortunately, prospective studies are hard to conduct in real life, and it may be difficult to compare results. A prospective study was presented by Wang et al. [47], where 58 students evaluated bug localization on fixed bugs in SWT.

6 Conclusion and Future Work

To be able to help locate a bug, a supporting system needs to find most likely files among numerous source code files in a project [14]. Algorithms based on learning–to–rank show promising results [3], [12], [13], [16], often outperforming other approaches. In this study we propose a generalized view on a building block for such algorithms in the context of bug localization. Then, we propose a new adaptive algorithm based on the pointwise approach. We propose a new initial ranking scheme that facilitates the construction of a more robust training target function, which eventually allows successfully applying the pointwise learning to rank algorithm. Our algorithm is designed in such a way that it will adapt its parameters to the characteristics of a project, without the need for separate parameters fitting procedures, as described in Section 3. Furthermore, it will adapt over time to changes in project characteristics (see Section 4.3.6 for selected parameters). The experimental evaluations on two datasets shows that the proposed adaptive approaches can be competitive compared to the recent state–of–the–art.

There are several possible future work directions. Other scoring functions and regression models can be examined. There is also room for improvements in the feature engineering, where other features could be constructed. For example one can use the cyclomatic complexity, or natural language processing techniques such as n-grams. Some potential lays in exploring how a complex training target function can impact setups based on pairwise and listwise learning to rank approaches. Also, there is a viable need for a more diversified dataset that would include variety of projects, with different bug handling policies, and mixed programming languages. Additionally, we hope to extend our bug localization algorithm to work on method/code block level for better granularity.

As a final note, we strongly believe that conducting replicable research in bug localization is the right direction. Publishing source code, results and datasets will not only simplify the replication, and evaluation of new algorithms, it will also fast-forward the adaptation in real life scenarios.

References

[1] S. Garfinkel. (2005) History's worst software bugs.

[2] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *SCIENCE CHINA Information Sciences*, vol. 58, no. 2, 2015.

[3] X. Ye, R. C. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014.

[4] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013.

[5] G. Antoniol and Y. Guéhéneuc, "Feature identification: A novel approach and a case study," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 2005, Budapest, Hungary*. IEEE Computer Society, 2005.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994.

[7] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 2008*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008.

[8] A. T. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, 2011*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds. IEEE Computer Society, 2011.

[9] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, 2012*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012.

[10] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013.

[11] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 2014*. IEEE Computer Society, 2014.

[12] X. Ye, R. C. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Trans. Software Eng.*, vol. 42, no. 4, 2016.

[13] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," *Appl. Soft Comput.*, vol. 62, 2018.

[14] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance*, vol. 19, no. 2, 2007.

[15] S. Wang and D. Lo, "Version history, similar report, and structure: putting them together for improved bug localization," in *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, 2014*, C. K. Roy, A. Begel, and L. Moonen, Eds. ACM, 2014.

[16] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, 2016.

[17] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information & Software Technology*, vol. 82, 2017.

[18] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE Trans. Software Eng.*, vol. 39, no. 11, 2013.

[19] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015.

[20] T. Dilshener, M. Wermelinger, and Y. Yu, "Locating bugs without looking back," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016.

[21] X. Huo, M. Li, and Z. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, S. Kambhampati, Ed. IJCAI/AAAI Press, 2016.

[22] V. Dang, "The Lemur Project - Wiki - Ranklib," https://sourceforge.net/p/lemur/wiki/RankLib/, The Lemur Project, [Online].

[23] L. Moreno, G. Bavota, S. Haiduc, M. D. Penta, R. Oliveto, B. Russo, and A. Marcus, "Query-based configuration of text retrieval solutions for software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015.

[24] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, 2011.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[26] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[27] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[28] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv preprint cs/0205028*, 2002.

[29] T. Joachims, "Training linear svms in linear time," in *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 2006*, T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, Eds. ACM, 2006.

[30] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, 2003.

[31] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, 1952.

[32] M. S. Nash, "Handbook of parametric and nonparametric statistical procedures," *Technometrics*, vol. 43, no. 3, 2001.

[33] H. Levene, "Contributions to probability and statistics," *Essays in honor of Harold Hotelling*, 1960.

[34] T. Hastie, S. Rosset, J. Zhu, and H. Zou, "Multi-class adaboost," *Statistics and its Interface*, vol. 2, no. 3, 2009.

[35] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings*, ser. Lecture Notes in Computer Science, P. M. B. Vitányi, Ed., vol. 904. Springer, 1995.

[36] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, 2006.

[37] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*, ser. Springer series in statistics. Springer, 2009.

[38] B. C. Ross, "Mutual information between discrete and continuous data sets," *PloS one*, vol. 9, no. 2, 2014.

[39] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, ser. ACM International Conference Proceeding Series, C. E. Brodley, Ed., vol. 69. ACM, 2004.

[40] A. B. Owen, "A robust hybrid of lasso and ridge regression," *Contemporary Mathematics*, vol. 443, no. 7, 2007.

[41] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, 2004.

[42] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007.

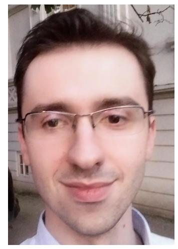
[43] K. Meffert, "The JGAP library," https://sourceforge.net/projects/jgap/, [Online].

[44] H. Hofmann, K. Kafadar, and H. Wickham, "value plots: Boxplots for large data," *Technical report, had. co. nz*, 2011.

[45] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: do they matter?" in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014.

[46] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, 2018*. IEEE Computer Society, 2018.

[47] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015.



Mikołaj Fejzer received the BSc and MS degrees in computer science from the Faculty of Mathematics and Computer Science, Nicolaus Copernicus University in Toruń. He is currently working toward the PhD degree. His research interests include mining software repositories, machine learning and software engineering.

Jakub Narębski studied at Inter-faculty Studies in Mathematics and Natural Sciences, and received the MSc and PhD degrees in physics from the University of Warsaw, Poland. His scientific interests include machine learning, GPGPU and scientific computations. He currently works at the Faculty of Mathematics and Computer Science of the Nicolaus Copernicus University in Toruń. Author of "Mastering Git" book published by Packt.

Piotr Przymus obtained his MSc computer science from Nicolaus Copernicus University in Toruń, Poland and his PhD in computer science from University of Warsaw, Poland. His main scientific interests are machine learning, data mining, time-series, GPGPU and distributed computations. Currently he works at Nicolaus Copernicus University in Toruń. He did his postdoc at LiF Aix-Marseille University, France.

Krzysztof Stencel received the MSc and PhD degrees in computer science from the University of Warsaw, Poland. He then received the DSc degree in computer engineering from Institute of Institute of Computer Science, Polish Academy of Sciences. He currently is a full professor at the University of Warsaw, Poland. He has published papers in databases and software engineering. He spent one year (2012-2013) as a visiting professor at Google.