Empir Software Eng (2017) 22:2237–2279
DOI 10.1007/s10664-016-9484-y

Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools

Tien-Duy B. Le[1] · Ferdian Thung[1] · David Lo[1]

Published online: 8 December 2016

Abstract Information retrieval (IR) based bug localization approaches process a textual bug report and a collection of source code files to find buggy files. They output a ranked list of files sorted by their likelihood to contain the bug. Recently, several IR-based bug localization tools have been proposed. However, there are no perfect tools that can successfully localize faults within a few number of most suspicious program elements for every single input bug report. Therefore, it is difficult for developers to decide which tool would be effective for a given bug report. Furthermore, for some bug reports, no bug localization tools would be useful. Even a state-of-the-art bug localization tool outputs many ranked lists where buggy files appear very low in the lists. This potentially causes developers to distrust bug localization tools. In this work, we build an oracle that can automatically predict whether a ranked list produced by an IR-based bug localization tool is likely to be effective or not. We consider a ranked list to be effective if a buggy file appears in the top-N position of the list. If a ranked list is unlikely to be effective, developers do not need to waste time in checking the recommended files one by one. In such cases, it is better for developers to use traditional debugging methods or request for further information to localize bugs. To build this oracle, our approach extracts features that can be divided into four categories: score features, textual features, topic model features, and metadata features. We build a separate prediction model for each category, and combine them to create a composite prediction model which is used as the oracle. We name this solution APRILE, which

Communicated by: Lin Tan

✉ Tien-Duy B. Le
btdle.2012@smu.edu.sg

Ferdian Thung
ferdiant.2013@smu.edu.sg

David Lo
davidlo@smu.edu.sg

[1] School of Information Systems, Singapore Management University, Singapore, Singapore

stands for Automated PRediction of IR-based Bug Localization's Effectiveness. We further integrate APRILE with two other components that are learned using our bagging-based ensemble classification (BEC) method. We refer to the extension of APRILE as APRILE$^+$. We have evaluated APRILE$^+$ to predict the effectiveness of three state-of-the-art IR-based bug localization tools on more than three thousands bug reports from AspectJ, Eclipse, SWT, and Tomcat. APRILE$^+$ can achieve an average precision, recall, and F-measure of 77.61 %, 88.94 %, and 82.09 %, respectively. Furthermore, APRILE$^+$ outperforms a baseline approach by Le and Lo and APRILE by up to a 17.43 % and 10.51 % increase in F-measure respectively.

Keywords Text classification · Information retrieval · Bug reports · Bug localization · Effectiveness prediction

1 Introduction

Software debugging is an important task to maintain software quality. However, debugging is an expensive task as it requires much time and manual labor to find root causes of bugs and correctly fix them. In 2002, software bugs are reported to cost US economy more than 50 billion dollars annually (Tassey 2002). Therefore, there are demands to develop tools which make debugging less costly.

To reduce debugging cost, several techniques have been proposed to support developers in locating the root cause of bugs. One family of techniques is referred to as information retrieval (IR) based bug localization techniques (Zhou et al. 2012; Saha et al. 2013; Wang and Lo 2014). An IR-based bug localization technique takes as input a textual bug report and a collection of source code files. It outputs a ranked list of files sorted by their suspiciousness scores which are computed by considering the textual similarity of the files to the input bug report. This ranked list of files is then forwarded to developers for manual inspection. Developers can inspect the files one-by-one starting from the most suspicious to the least suspicious ones.

Recently, Kochhar et al. have surveyed hundreds of developers from more than 30 countries on how they perceive fault localization (Kochhar et al. 2016). Their study highlights that a large majority of the developers value research on fault localization and would like to use fault localization tools if they meet some criteria. One of the criteria is the *trustworthiness* of a fault localization tool, which is addressed in this work. Moreover, Xia et al. have conducted a study on a fault localization tool and find that using a *trustworthy* tool, developers can significantly reduce the cost involved in localizing a fault (Xia et al. 2016). Imagine bug reports that are automatically populated with a list of relevant files by an IR-based bug localization tool. In order to help developers, start and finish them more quickly, it would be crucial to know whether the automatically generated results were reliable or not. If an IR-based bug localization tool is effective, developers should be able to find a buggy file by inspecting just a few files at the top of the ranked list. Unfortunately, the current state-of-the-art IR-based bug localization tools are far from perfect. Currently, there is *no* perfect bug localization tool that can successfully localize faults within a few number of most suspicious program elements for every single input bug report. It is unclear whether there will ever be such a tool in the future. These mean that there will be bug reports where a more advanced bug localization tool is bad (i.e., ineffective), but a less advanced one is good at these bug reports. With the many IR-based bug localization tools proposed in the

literature, e.g., Zhou et al. (2012); Saha et al. (2013); Wang and Lo (2014), etc., it is difficult for developers to decide which tool would be effective for a given bug report. Moreover, for some bug reports, no bug localization tools would be useful. Developers would waste time to go through the output of bug localization tools. Recently, Parnin and Orso conducted a user study on an automatic debugging tool, and find that developers do not find an automatic debugging tool useful if they cannot find the root cause of a bug early in a ranked list (Parnin and Orso 2011). These ineffective cases can make developers lose confidence in bug localization tools. A similar observation was made for many static analysis bug finding tools (e.g., FindBugs (Hovemeyer and Pugh 2004) and Lint (Johnson 1978)) that typically return many false positive warnings, which potentially cause developers to distrust their outputs (Heckman and Williams 2011; Kim and Ernst 2007; Johnson et al. 2013; Ayewah and Pugh 2010).

In this work, we mitigate the impact of the unreliability of IR-based bug localization tools by proposing a prediction framework that is able to compute the likelihood whether an output of a bug localization tool is effective or not. We consider the output of a bug localization tool as effective if a buggy file is among the top-N files. With the help of our approach, developers can decide whether they want to use the output of a bug localization tool or not. If the output of a bug localization tool is unlikely to be effective, developers are better off to use traditional debugging methods to find the bug.

Our approach can potentially be integrated to an IDE and a bug tracking system. Given a new bug report to be debugged in a bug tracking system (e.g., JIRA or Bugzilla), our approach would run a number of IR-based bug localization techniques in the background, and predicts their effectiveness. It will then pick the bug localization tool's output that is most likely to be effective, and highlight potentially buggy files to developers in the IDE. If none of the outputs are likely to be successful, our approach will notify the developer that there is no good recommendation and the developer can either improve the description of the bug report or proceed with traditional debugging. Our approach can also be used standalone. For this setting, developers can directly input the directory containing the source code files, and the text in a bug report that he/she wants to debug, and our approach can process this input to produce a list of potentially buggy files, or declare that no good recommendation can be made.

To predict the effectiveness of a bug localization instance, i.e., the application of a bug localization tool on a bug report, we extract important features from the input bug report and the suspiciousness scores that are output by the bug localization tool. The set of extracted features can be divided into four categories: features extracted from suspiciousness scores, features extracted from words that appear in the textual contents of a bug report, features extracted from topic models learned from the textual contents of a bug report, and features extracted from the metadata of a bug report. For each feature category subset, we learn a separate prediction model from a training dataset using a machine learning technique (i.e., Support Vector Machine). A prediction model outputs a score given a bug localization instance indicating the likelihood that the instance is effective. From the resultant four models, one for each feature category, we create a final prediction model which combines the four models by computing a weighted sum of their prediction scores. The weights are tuned to maximize the prediction result on the training dataset. Then, we use the final prediction model to predict the effectiveness of bug localization instances whose effectiveness are unknown. We name this solution APRILE (Automated PRediction of IR-based Bug Localization's Effectiveness), which was introduced in our conference paper (Le et al. 2014b).

In this journal paper, we propose an extension of APRILE and call it APRILE$^+$. Intuitively, a combination of several prediction approaches and a voting scheme generally outperforms a single learner (Bauer and Kohavi 1999; Breiman 1996a; Lemmens and Croux 2006; Prasad et al. 2006). Therefore, APRILE$^+$ extends APRILE by integrating APRILE with two other components that are learned using our bagging-based ensemble classification (BEC) method. Each of these three components will output a recommendation and APRILE$^+$ will output a final recommendation based on *majority voting*, i.e., the output (effective or ineffective) that is recommended by at least two out of the three components will be the final prediction output.

We have evaluated the performance of APRILE$^+$ to predict the effectiveness of state-of-the-art bug localization techniques (i.e., BugLocator (Zhou et al. 2012), BLUiR (Saha et al. 2013), and AmaLgam (Wang and Lo 2014)) which are applied on a dataset of 3,800 bugs from AspectJ, Eclipse, SWT, and Tomcat. Among these bugs, 3,459 bugs[1] from AspectJ, Eclipse, and SWT were used to evaluate state-of-the-art IR-based bug localization tools. Our experimental results show that APRILE$^+$ can achieve an average precision, recall, and F-measure of 77.61 %, 88.94 %, and 82.09 % with effectiveness criteria of $N = 10$, respectively.

In terms of F-measure (i.e., the harmonic mean of precision and recall), APRILE$^+$ outperforms a baseline based on the approach proposed by Le and Lo (Le and Lo 2013; Le et al. 2014a) by up to 17.43 %. Le and Lo proposed an approach that predicts the effectiveness of a spectrum-based bug localization tool, e.g., Tarantula (Jones and Harrold 2005). Spectrum-based bug localization tools analyze execution traces rather than bug reports. We adapt their approach to predict the effectiveness of an IR-based bug localization tool and use it as the baseline. We have also compared the performance of APRILE$^+$ against the performance of APRILE and find that APRILE$^+$ can outperform APRILE for all datasets by up to a 10.51 % increase in F-measure.

Compared to the existing body of work in machine learning, our approaches (APRILE$^+$ and APRILE) are novel in the following aspects:

(i) Although bagging based methods are quite popular, they have not been used for predicting effectiveness of bug localization instances. We are the first to apply a bagging based method to predict effectiveness of IR-based bug localization instances.
(ii) Although individual parts of APRILE$^+$ are not novel, the composition is not novel. In fact, not all classification algorithms can be composed with APRILE, and result in improvement as we have achieved. We have investigated a number of compositions and find a working one.
(iii) The features that we extract to characterize IR-based bug localization instances are peculiar of the software engineering problem being tacked (i.e., the prediction of the effectiveness of IR-based bug localization tools).

The contributions of our work (inclusive of our preliminary conference paper) are as follows:

1. We propose a comprehensive list of features extracted from an input bug report and suspiciousness scores. Our features are divided into four categories: score, text, topic model, and metadata features.

[1] https://bugcenter.googlecode.com/files/BugLocator.zip

2. We propose a framework APRILE that utilizes all feature categories to predict the effectiveness of an IR-based bug localization tool. For each feature category, we create a separate prediction model trained by Support Vector Machine (SVM) algorithm. We then construct a final model which combines the four prediction models.
3. We propose an extended framework APRILE$^+$ which extends APRILE by combining it with two additional components learned using our Bagging-based Ensemble Classification (BEC) approach. The three models are combined by integrating the recommendations made by them using majority voting.
4. We evaluate our approach on a dataset of 3,800 bug reports from four software projects. The empirical results show that APRILE$^+$ performs well for various datasets and settings.

We organize the remainder of our paper as follows. In Section 2, we discuss background information on IR-based bug localization, and ensemble learning. Then, we present APRILE and APRILE$^+$ in Section 3. Next, Section 4 describes our experiment settings and results. We discuss related work in Section 6. Finally, we conclude and present future work in Section 7.

2 Background

In this section, we discuss some background materials on IR-based bug localization and ensemble learning.

2.1 IR-Based Bug Localization

An IR-based bug localization approach takes as input a textual bug report and a collection of program source code files. Its output is a ranked list of files sorted by their likelihood to be a buggy file that needs to be fixed to resolve the bug report. This ranked list is then manually inspected from the beginning until the buggy files are identified.

Recently, many IR-based bug localization tools have been proposed (Zhou et al. 2012; Saha et al. 2013; Wang and Lo 2014). The main idea behind an IR-based bug localization tool is that a bug report and buggy files are likely to share common words. Also, if a program file has higher textual similarity to the bug report than other files, it is more likely to contain the bug. By deploying text retrieval models, IR-based bug localization tools calculate similarity scores between a bug report and program files. Next, program files are sorted in descending order of their textual similarity scores, and forwarded to developers for manual investigation.

IR-based bug localization first extracts text that appear in summary and description fields of bug reports. It also extracts comments and identifiers that appear in source code files. Each bug report and source code file can then be represented by a textual document. Next these documents are input to a text preprocessing procedure which consists of three main steps: text normalization, stopword removal, and stemming. These steps are described below:

– *Text Normalization:* In this step, special symbols and punctuation marks are removed from a document. Next, the document is split into its constituent words. If a word is an identifier in a source code file, it is again split into smaller words following the Camel casing convention (e.g., “processFile” is split into “process” and “file”).

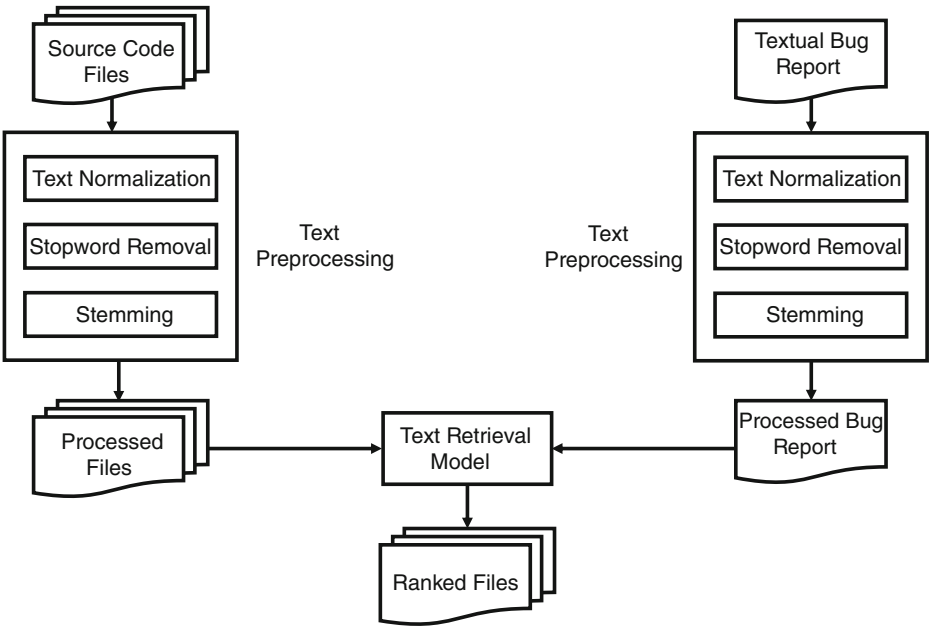

Fig. 1 Information Retrieval based Bug Localization Framework

- *Stopword Removal:* In this step, we remove English stopwords[2] from the normalized document. These stopwords frequently appear in many documents and do not help much in differentiating one document from another.
- *Stemming:* In this step, words are transformed to their root forms. For example, "localized", "localization", "localize", and "locally" are all simplified to "local".

After the above steps are performed, the document representing a bug report is then compared to documents representing source code files using various text retrieval models. These text retrieval models assign weights to words in the documents and, based on common words and their weights, compute similarity of one document and another. The details of the text retrieval model differ for different bug localization techniques. Figure 1 shows the process of generating a ranked list of source code files for an input bug report.

The state-of-the-art approaches are BugLocator proposed by Zhou et al. (2012), BLUiR proposed by Saha et al. (2013), and AmaLgam proposed by Wang and Lo (2014). We describe the details of these approaches below.

BugLocator BugLocator (Zhou et al. 2012) processes a bug report and produces a ranked list of candidate buggy files based on several heuristics. The first heuristic is a bug report and its relevant buggy files are likely to share similar words. The second heuristic is larger files are more likely to be buggy. The third heuristic is two bug reports that are similar to each other (in terms of their textual descriptions) are likely to be mapped to similar buggy files.

[2] http://dev.mysql.com/doc/refman/5.1/en/fulltext-stopwords.html

To make use of the first and second heuristics, BugLocator compares the pre-processed words in the bug report with the pre-processed words in each source code file. The comparison is performed using a *revised* vector space model (rVSM) which is an extension of the standard vector space model (VSM). A vector space model (VSM) represents words in a bug report as a vector, and words in a source code file as another vector. The similarity of the bug report and the source code file is computed by taking the *cosine similarity* of the two vectors, which is the dot product of the two vectors normalized by the product of the two vectors' magnitudes (Manning et al. 2008). rVSM extends VSM by multiplying the cosine similarity score with a weight that is larger for larger source code files to produce the *rVSMScore* – this is based on the second heuristic.

To make use of the third heuristic, BugLocator compares a new bug report against historical bug reports that have been fixed earlier. BugLocator then identifies files that are fixed to resolve these similar past historical bug reports and computes their likelihood to be the relevant buggy files of the new bug report (*SimiScore*). The *rVSMScore* and the *SimiScore* are then combined together to produce a final score by taking their weighted sum. Source code files are then sorted based on their final scores and the ranked list of files are presented to developers to help them in debugging.

BLUiR BLUiR (Saha et al. 2013), proposed by Saha et al., improves the performance of BugLocator by making use of *structured* information retrieval. Rather than treating each bug report as one textual document, it splits the bug report into several parts: summary and description. Each part corresponds to a field in a bug report – summary refers to a few-word text that outlines a bug report, while description refers to a few-sentence text that elaborates the symptoms of the bug and/or conditions that lead to failures that are caused by the bug. Also, BLUiR breaks a source code file into several parts: class, method, variable, and comments. These parts correspond to words that appear in class names (class), method names (method), variable names (variable), and comments (comments). To compute the similarity of a bug report to a source code file, BLUiR compares the similarity of each part of the bug report to each part of the source code file. These similarity scores are summed up together to produce the final score. Saha et al. have shown that BLUiR outperforms BugLocator.

AmaLgam AmaLgam (Wang and Lo 2014), proposed by Wang and Lo, also uses structured information retrieval like BLUiR, however, it considers additional heuristics that are not considered by BLUiR. Similar like BugLocator, AmaLgam also compares a new bug report to historical bug reports that have been fixed before. Additionally, AmaLgam considers information gleaned from the version control system to identify files that are more likely to be buggy. Files that are buggy many times before are likely to continue to be buggy in the future. AmaLgam amalgamates these heuristics to create a more effective bug localization technique. Wang and Lo have shown that AmaLgam outperforms BLUiR.

2.2 Bagging Method

Bagging (Breiman 1996b), which stands for bootstrap aggregation, is a method to improve accuracy and stability of classification models (i.e., classifiers). Despite of its simplicity, bagging is an effective method for ensemble learning (i.e., combining multiple classification models) as it reduces variance and avoids overfitting. Originally, the method is proposed to work with tree-based models, but it is applicable to other classification models. Given a training data, bagging method creates several samples (with replacement). Each sample

is referred to as a bootstrap sample, and used for constructing a base classification model. Subsequently, the output of these models are combined by voting to return one final output.

3 Proposed Approach

In this section, we first describe information of extracted features in Section 3.1. Then, Section 3.2 discusses technical aspects of APRILE^{+}.

3.1 Feature Extraction

In this section, we list features that we extract from a bug localization instance. These include features that we extract from suspiciousness scores produced by a bug localization tool (score features), features that we extract from words that appear in textual contents of an input bug report (text features), features that we extract from topic distributions of the bug report (topic model features), and features that we extract from metadata of the bug report (metadata features). The following sub-sections describe these features one by one.

3.1.1 Score Features

Table 1 shows a list of features that we extract from the suspiciousness scores. A bug localization tool outputs a suspiciousness score for every source code file. In the table, features

Table 1 List of Score Features

ID	Description
Raw Scores (20 features)	
R_1	Highest suspiciousness score
R_2	Second highest suspiciousness score
R_i	i^{th} highest suspiciousness score ($3 \leq i \leq 20$)
Simple Statistics of Raw Scores (5 features)	
SS_1	Mean of $\{R_1, \ldots, R_{20}\}$
SS_2	Median of $\{R_1, \ldots, R_{20}\}$
SS_3	Mode of $\{R_1, \ldots, R_{20}\}$
SS_4	Variance of $\{R_1, \ldots, R_{20}\}$
SS_5	Standard deviation of $\{R_1, \ldots, R_{20}\}$
Gaps (21 features)	
G_1	$R_1 - R_2$
G_2	$R_2 - R_3$
G_i	$R_i - R_{i+1} (3 \leq i < 20)$
G_{min}	Min of $\{G_1, \ldots, G_{19}\}$
G_{max}	Max of $\{G_1, \ldots, G_{19}\}$
Relative Differences (18 features)	
RD_1	$\frac{R_2 - R_{20}}{R_1 - R_{20}}$
RD_2	$\frac{R_3 - R_{20}}{R_1 - R_{20}}$
RD_i	$\frac{R_{i+1} - R_{20}}{R_1 - R_{20}}$ $(3 \leq i \leq 18)$

R_1 to R_{20} are the suspiciousness scores of the top-20 files in a ranked list. If the suspiciousness scores of top ranked files are relatively low, then the instance is likely to be ineffective. The next five features, SS_1 to SS_5, are simple statistics of the top-20 suspiciousness scores. Next, features G_1 to G_{19}, G_{min}, and G_{max} capture absolute differences between two consecutive suspiciousness scores. If the difference between two consecutive scores are large, the corresponding two files are very different from each other. An effective bug localization instance should be able to separate buggy files from other files. Finally, the last 18 features, RD_1 to RD_{18}, capture how diverse the values of R_1 to R_{20} are. The suspiciousness scores of top-ranked files corresponding to an effective bug localization instance are likely to be diverse. If all files are given the same suspiciousness score, then the bug localization instance will be ineffective.

3.1.2 Text Features

We extract text from the summary and description fields of a bug report, and use them as text features of a bug localization instance. Table 2 shows a list of features that we extract from the textual contents of a bug report. The first feature in our list (TRACE) has a boolean value. Its value is 1 if the summary or description field of a bug report contains a stack trace. Otherwise, its value is 0. A program stack trace usually contains clues leading to a buggy file as it contains names of relevant program files (e.g., "at org.aspectj.EclipseFactory.fromBinding(EclipseFactory.java:202)"). Hence, the existence of a program stack trace can help an IR-based bug localization tool to effectively localize bugs. Furthermore, we consider each word in a bug report as a feature, and its value is the number of times the word appears in the bug report. Before selecting words from bug reports as features, we perform text preprocessing that is described in Section 2.1.

3.1.3 Topic Model Features

Topic modeling is a technique to discover latent topics in a collection of documents. These latent topics are inferred based on the occurrences of words in the documents. One of the most popular topic modeling techniques is Latent Dirichlet Allocation (LDA) (Blei et al. 2003). LDA posits that each document is a mixture of topics and each word in the document is associated to a topic. Given a document, LDA generates its topic distribution, which corresponds to the probability of each topic to be assigned to the document. We apply Latent

Table 2 List of text features

ID	Description
TRACE	One, if there is a stack trace in the summary or description fields of a bug report. Zero, otherwise.
F_{w_1}	Number of times word w_1 occurs in the summary and description fields of a bug report.
...	
F_{w_i}	Number of times word w_i occurs in the summary and description fields of a bug report.
...	
F_{w_n}	Number of times word w_n occurs in the summary and description fields of a bug report.

Dirichlet Allocation (LDA) (Blei et al. 2003) to extract a number of features from bug reports. LDA accepts a number of input parameters:

1. k, which is the number of topics that should be inferred from the input documents (i.e., bug reports).
2. α, which affects the topic distributions per documents. Higher values of α make topics more uniformly distributed (i.e., better smoothing of topics) in each document.
3. β, which governs the word's distributions per topics. Higher values of β lead to words more uniformly distributed in every topic.
4. n, which is the number of Gibbs's iterations i.e., the number of times Gibbs sampler is invoked (Blei et al. 2003).

The output of applying LDA is a topic-model that contains the following information:

1. k topics, where each topic is a distribution of words.
2. Probability of topic t to occur in bug report br.
3. Topic assigned to word w in bug report br.

Each time LDA is applied, it creates a topic model M_k where k is the number of topics. For our approach, we apply LDA with $k \in \{5, 10, 15\}$ to infer three topic model M_5, M_{10}, M_{15}. Then, we capture interesting features from the three topic modes. Importantly, LDA models can be used to estimate topic probabilities of an unseen bug report that does not belong to the training corpus (Blei et al. 2003). Therefore, in deployment phase, we calculate topic features (i.e., topic probabilities) of emerging bug reports without updating M_5, M_{10}, and M_{15} models. However, new bug reports might have hidden topics that one single topic model cannot capture. For that reason, we construct a number of different topic models (i.e., M_5, M_{10}, and M_{15}) in training phase to maximize the coverage on hidden topics of new bug reports. Table 3 lists features that we extract from these models.

For each topic model M_k ($k \in \{5, 10, 15\}$), we use the topic probabilities of a bug report as the features of the corresponding bug localization instance. Each topic is an abstraction of a set of words. The set of topics inferred by a topic modeling technique based on the number of topics setting k represents the level of abstraction. The higher the value of k is,

Table 3 List of topic model features. M_k is a topic model with the number of topics set to k ($k \in \{5, 10, 15\}$)

ID	Description
Raw Topic Probabilities	
TM_k^1	Probability of the 1^{st} topic appearing in M_k
TM_k^2	Probability of the 2^{nd} topic appearing in M_k
TM_k^i	Probability of the i^{th} topic ($3 \leq i \leq k$) appearing in M_k
Simple Statistics of Topic Probabilities in M_k	
TS_k^1	Max of $\{TM_k^1, \ldots, TM_k^i\}$
TS_k^2	Median of $\{TM_k^1, \ldots, TM_k^i\}$
TS_k^3	Variance of $\{TM_k^1, \ldots, TM_k^i\}$
TS_k^4	Standard deviation of $\{TM_k^1, \ldots, TM_k^i\}$
TS_{avg}^1	$(TS_5^1 + TS_{10}^1 + TS_{15}^1)/3$

the lower the abstraction level is. By using multiple topic models with various k values, we capture information from many different abstraction levels in order to maximize the chance to differentiate bug reports corresponding to effective bug localization instances from other reports that lead to ineffective instances. In addition to the raw topic probabilities, we also compute simple statistics of these probabilities as features.

3.1.4 Metadata Features

In addition to summary and description fields, bug reports have other fields. These fields provide basic information such as report date, severity, priority, etc. We refer to this information as metadata of bug reports.

Table 4 shows a list of metadata features that we are interested in. In total, there are 14 metadata features. Among the features, features MT_1 and MT_2 capture severity and priority of the reported bug. Intuitively, bug reports assigned with high severity or priority are likely typically to be highly-noticeable bugs that affect major functionalities of an application. These bugs are often well described by reporters, and thus the bug reports are likely to contain important and highly relevant words that can better lead IR-based bug localization tools to the exact faulty files. Next, features MT_3 to MT_7 capture the context where the bug is observed, and features MT_8 to MT_{14} capture factors that might impact the quality of bug reports such as the experience of the reporter, the number of attachments, etc. We utilize features extracted from bug report metadata to maximize the chance to capture distinctive characteristics of bug reports that correspond to effective bug localization instances.

Table 4 List of Metadata Features

ID	Description
Importance of a Bug	
MT_1	Priority
MT_2	Severity
Context of a Bug	
MT_3	Product
MT_4	Component
MT_5	Software version
MT_6	Hardware platform (e.g., PC etc.)
MT_7	Operating system platform (e.g., XP, Linux etc.)
Quality of a Bug Report	
MT_8	Name of reporter
MT_9	Number of bug reports the reporter has submitted so far at time of the current bug report.
MT_{10}	Number of persons in CC list when the bug report is first submitted.
MT_{11}	Total number of attachments in the bug report
MT_{12}	Number of attachments that are applications
MT_{13}	Number of attachments that are texts
MT_{14}	Number of attachments that are images

3.2 APRILE+

In this section, we first describe the overall framework of APRILE+. Next, we present the details of APRILE (Le et al. 2014b)'s effectiveness prediction model and *bagging-based ensemble classification* (BEC) method that are used to build several components of APRILE+.

3.2.1 Overall Framework

Figure 2 shows the overall framework of APRILE+. There are two main phases in the framework: *training* and *deployment*. In the training phase, APRILE+ takes as input a set of training bug localization instances and their corresponding effectiveness labels. Each instance in the training set corresponds to a bug and comes with the following information:

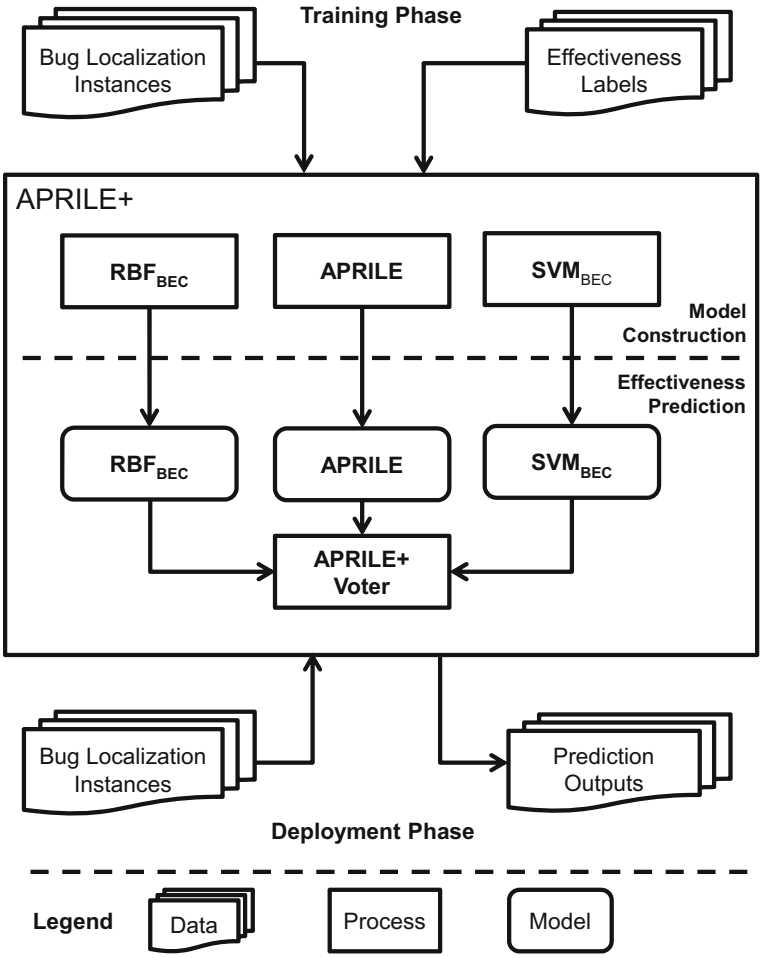

Fig. 2 APRILE+ Framework

1. A bug report that describes the bug.
2. Suspiciousness scores that are assigned to source code files by an IR-based bug localization tool.
3. Effectiveness label (i.e., "effective" or "ineffective").

These inputs are then used to construct the three classifiers inside APRILE$^+$ (i.e., APRILE, SVM_{BEC}, and RBF_{BEC}). Each of these classifiers has its own training phase where features are first extracted from the inputs (see Section 3.1), and prediction models are constructed according to its learning strategy (see Section 3.2.2 and Section 3.2.3). In the deployment phase, APRILE$^+$ takes as input the three learned classifiers and a set of bug localization instances whose effectiveness is to be predicted. APRILE$^+$ applies the classifiers on each of the bug localization instances and receives a prediction output (which specifies whether the instance is likely to be effective or not) from each classifier. APRILE$^+$ Voter takes these prediction outputs and employs *majority voting* to pick the final effectiveness label (i.e., the effectiveness label which is predicted by the majority of the classifiers is considered as the final classification label). To avoid a tie in the voting process, APRILE$^+$ contains an odd number of classifiers in its framework.

APRILE$^+$ is the combination of RBF_{BEC}, SVM_{BEC}, and APRILE (Le et al. 2014b). Among the three classifiers, RBF_{BEC} and SVM_{BEC} are trained from the set of features presented in Section 3.1 (i.e., score features, text features, topic features and metadata features) by the bagging-based ensemble classification (BEC) method (see Section 3.2.3). Compared to APRILE, the BEC method is generic and can be used to build many other additional classifiers by using one of the off-the-shelf classification algorithms (e.g., k-nearest neighbor, random forest, etc.) as the underlying (or base) classification algorithm. From the extracted features, we respectively apply BEC to construct RBF_{BEC} and SVM_{BEC} models with RBF network and SVM as underlying classification algorithms.

A radial basis function (RBF) network is a type of artificial neural network that can be used for supervised learning problems including classification (Broomhead and Lowe 1988; Mitchell 1997). The goal of a RBF network is to learn to convert a set of inputs into an output given a set of labeled examples. A RBF network consists of several layers, namely the input layer, hidden layer, and output layer, where each layer consists of a set of nodes. The nodes (or neurons) in the input layer corresponds to the set of inputs. The nodes in the hidden layer implements a set of radial basis functions (i.e., Gaussian functions) which convert a set of inputs into intermediary outputs. The node in the output layer sums up the intermediary outputs generated by the hidden layer. In the training process, the weights of edges connecting nodes in the input layer to nodes in the hidden layer are first determined. Next, the weights of edges connecting nodes in the hidden layer to nodes in the output layer are determined. RBF network can be trained in a short amount of time and it has good performance for various classification problems.

SVM is a popular classification algorithm that has been shown effective for many kinds of problems (Han and Kamber 2006). It represents data instances as points in a multi-dimensional space where each feature is a dimension. It then separates data instances from different classes by finding a multi-dimensional hyperplane that best separates them. This hyperplane is often called the *maximum marginal hyperplane* (MMH). The underlying function (i.e., kernel) that defines the plane itself can be customized. The commonly used one is the linear kernel. In this work, we use SVM with linear kernel to build prediction models and consider a bug localization instance as a point in a multi-dimensional space. SVM is used to find the plane that separates effective bug localization instances from ineffective ones.

3.2.2 APRILE's Effectiveness Prediction model

APRILE's prediction model contains four internal components which analyze score, text, topic model, and metadata features. Each component consists of a prediction model that specializes in a particular feature category. For example, score component only analyzes the score features, text component only analyzes the text features, and so on. We use Support Vector Machine (SVM) to train the prediction model of each component.

In the training phase, SVM algorithm takes as input features of training bug localization instances whose effectiveness are known and learns a prediction model. A prediction model can process features of a bug localization instance and output a prediction score of that instance. The prediction score indicates how likely a bug localization instance is effective. If the score is greater than a threshold, then the corresponding instance is predicted as effective by that prediction model. Otherwise, it is predicted as ineffective. We linearly combine scores generated by the prediction models of the four internal components together to obtain the final score and prediction (i.e., APRILE score) as follows:

$$\text{APRILE}^{\text{PREDICTION}}(p) = \begin{cases} p \text{ is effective} & \text{if APRILE}(p) > \omega \\ p \text{ is ineffective} & \text{if APRILE}(p) \leq \omega \end{cases} \quad (1)$$

$$\begin{aligned} \text{APRILE}(p) = \alpha \times \text{SVM}_{\text{Score}}(p) \quad & +\beta \times \text{SVM}_{\text{Text}}(p) + \gamma \times \text{SVM}_{\text{Topic}}(p) \\ & +\delta \times \text{SVM}_{\text{Meta}}(p) \\ & (\alpha, \beta, \gamma, \delta \in [0, 1] \wedge \alpha + \beta + \gamma + \delta = 1) \end{aligned} \quad (2)$$

In the above equation, p is an input bug localization instance, $\text{APRILE}^{\text{PREDICTION}}(p)$ is the predicted effectiveness of p, $\text{APRILE}(p)$ is the combined prediction score for p, $\text{SVM}_{\text{Score}}(p)$, $\text{SVM}_{\text{Text}}(p)$, $\text{SVM}_{\text{Topic}}(p)$, and $\text{SVM}_{\text{Meta}}(p)$ are the prediction scores output by the score, text, topic model, and metadata components, respectively. Each of the component has a weight in range of [0, 1] and their sum equals to 1. We also define a threshold for APRILE to differentiate prediction scores of effective and ineffective instances. We denote the threshold as ω. If $\text{APRILE}(p) > \omega$, then p is an effective instance. Otherwise, p is ineffective.

We need to tune values of α, β, γ, δ and ω. We tune these values such that the performance of APRILE is maximized on the training data. We measure the performance of APRILE in terms of F-measure (see Section 4.1). We try different weight combinations by varying the value of each weight from 0 to 1, in a step of 0.025, with a constraint that the four weights will add up to 1. Implementation-wise, we pick the values of α, β, and γ; if their total weight is less than 1, we set the value of δ such that the summation of the four weights is equal to 1. Next, we follow (2) to calculate APRILE scores of bug localization instances . Then, we call procedure *tuneOmega* to tune the threshold ω such that the F-measure is maximized. We select the weight combination of α, β, γ, δ and ω that results in the best F-measure.

Next, we describe procedure *tuneOmega*. The procedure takes as input a set of bug localization instances, and their effectiveness labels. *tuneOmega* first sorts bug localization instances in ascending order of their APRILE scores (line 1). Next, *tuneOmega* calculates F-measures for two base cases. The first case is when all instances are predicted as effective (line 3). The second case is when all instances are predicted as ineffective (line 5). From

lines 11 to 17, *tuneOmega* iterates through various ω values by taking the average of the APRILE scores of two consecutive instances in the sorted list. The ω value which results in the highest F-measure is selected and returned.

Procedure tuneOmega

Input: D: Training set of IR-based bug localization instances
L: Effectiveness labels of instances in D
Output: Pair of best ω and best F-measure

```
1  Sort instances in D in ascending order of their APRILE's scores.
2  ω1 ← APRILE(D[1]) − 10^−3
3  Fm1 ← F-measure when effectiveness threshold is ω1
4  ω2 ← APRILE(D[D.length]) + 10^−3
5  Fm2 ← F-measure when effectiveness threshold is ω2
6  if Fm1 > Fm2 then
7  |  Fm_best ← Fm1, ω_best ← ω1
8  else
9  |  Fm_best ← Fm2, ω_best ← ω2
10 end
11 for k ← 1 to D.length − 1 do
12 |  ω ← (APRILE(D[k]) + APRILE(D[k + 1]))/2
13 |  Fm_temp ← F-measure when effectiveness threshold is ω
14 |  if Fm_temp > Fm_best then
15 |  |  Fm_best ← Fm_temp, ω_best ← ω
16 |  end
17 end
18 return (ω_best, Fm_best)
```

In the deployment phase, we apply the learned final prediction model to predict the effectiveness of bug localization instances whose effectiveness are unknown. Given an input instance p, we calculate APRILE(p) (see (2)) by using prediction models $\text{SVM}_{\text{Score}}$, SVM_{Text}, $\text{SVM}_{\text{Topic}}$, SVM_{Meta}, and weights $\alpha, \beta, \gamma, \delta$, which are learned in the training phase. Then, we compare APRILE(p) with threshold ω, which is also learned in the training phase, and output the predicted effectiveness label of p (i.e., "effective", or "ineffective").

We divide the features into 4 categories as each category of features captures a specific characteristic of bug localization instances. Each of them is distinctive and different from the others. For example, score features capture numeric properties of the output suspiciousness scores, which are different from information captured by text features, topic model features, and metadata features. Therefore, each prediction model inferred from a category of features has its own prediction power leveraging on specific characteristics of bug localization instances. Equation 2 combines these models together to formulate APRILE by tuning each model's contribution (i.e., parameter) in order to maximize the accuracy. We do not mix all the features together and learn a single model as the many features may interfere with one another making it harder to learn a good discriminative model, or one group of features may dominate the rest in an unbalanced way, and in the end result in a poorer effectiveness.

3.2.3 Bagging-Based Ensemble Classification (BEC) Method

Our bagging-based ensemble classification (BEC) method has two main phases: training and deployment. In the training phase, BEC takes as input a set of IR-based bug localization

instances along with their effectiveness labels. BEC also takes as input an underlying classification algorithm (e.g., RBF or SVM) that will be called multiple times to construct an ensemble of classifiers. We denote the ensemble of classifiers built using BEC method with CLA as the underlying classification algorithm as CLA_{BEC}. The deployment phase employs the BEC classifier constructed in the training phase to predict the effectiveness labels of unseen bug localization instances.

Algorithm 1 Constructing a category-specific BEC prediction model (aka. classifier)

Input: T: Training instances
NC: Number of classifiers
SR: Sampling rate
CLA: Base classification algorithm
FC: Feature category
Output: $CLA_{BEC,FC}$: A category-specific BEC classifier

1 $T^+ \leftarrow$ set of effective instances from T
2 $T^- \leftarrow$ set of ineffective instances from T
3 $n \leftarrow SR \times \min(|\ T^+\ |, |\ T^-\ |)$
4 $CLA_{BEC,FC} \leftarrow \{\}$
5 **for** $k \in \{1 \ldots NC\}$ **do**
6 $\quad X_k^+ \leftarrow$ randomly select n instances from T^+
7 $\quad X_k^- \leftarrow$ randomly select n instances from T^-
8 $\quad X_k \leftarrow X_k^+ \cup X_k^-$
9 $\quad$ Run CLA on X_k to construct a simple classifier SC_{X_k}
10 $\quad CLA_{BEC,FC} \leftarrow CLA_{BEC,FC} \cup SC_{X_k}$
11 **end**
12 **return** $CLA_{BEC,FC}$

In the training phase, BEC first extracts features from a training set of bug localization instances, and divides the extracted features into four categories (see Section 3.1). For each feature category FC, we create a category-specific BEC classifier on a training set of bug localization instances T using Algorithm 1. The algorithm takes as input a set of training instances T, a number of simple classifiers in the resultant BEC classifier NC, a sampling rate SR, an underlying/base classification algorithm CLA, and a feature category FC.

At lines 1 to 3, Algorithm 1 computes the value of variable n which is the number of effective and ineffective instances to be sampled. The value of n is set to be the product of the sampling rate SR and the number of instances in the minority class (i.e., effective or ineffective class) in T. At lines 5 to 10, the algorithm learns NC simple classifiers from NC samples, each containing $2 \times n$ instances, with equal numbers of effective and ineffective instances. The algorithm takes each of the generated samples X_k to learn a simple classifier SC_{X_k} using the base classification algorithm CLA. At line 12, it returns a category-specific BEC classifier which is a collection of simple classifiers $\{BC_{X_k} | k \in \{1 \ldots NC\}\}$.

With the above steps, we can generate a powerful ensemble classifier which is a collection of simple classifiers trained by the base classification algorithm CLA. The sampled data X_k (line 8 in Algorithm 1) used to build the simple classifiers would be diverse. Therefore, the effectiveness of the corresponding simple classifiers for different kinds of bug localization instances vary from sample to sample. That gives the BEC prediction model, which is a collection of the simple classifiers, more chances to capture various aspects of different kinds of bug localization instances.

Algorithm 2 Computing category-specific BEC prediction score

Input: p: IR-based bug localization instance
$\text{CLA}_{\text{BEC,FT}}$: A category-specific BEC classifier
Output: $\text{CLA}_{\text{BEC,FT}}(p)$: Category-specific BEC prediction score for instance p

```
effective ← 0
ineffective ← 0
for SC_{X_k} ∈ CLA_{BEC,FT} do
    prediction ← Apply SC_{X_k} on p
    if prediction == "effective" then
        effective ← effective + 1
    else
        ineffective ← ineffective + 1
    end
end
return effective − ineffective
```

In total, we construct four category-specific BEC classifiers corresponding to the score, text, topic model, and metadata features. To create a more powerful prediction model (i.e., classifier), we construct a final prediction model based on the four classifiers as follows:

$$\begin{aligned}\text{CLA}_{\text{BEC}}(p) = \alpha \times \text{CLA}_{\text{BEC,Score}}(p) \; + \; \beta \times \text{CLA}_{\text{BEC,Text}}(p) \\ +\gamma \times \text{CLA}_{\text{BEC,Topic}}(p) \; + \; \delta \times \text{CLA}_{\text{BEC,Meta}}(p) \\ (\alpha, \beta, \gamma, \delta \in [0, 1] \; \wedge \; \alpha + \beta + \gamma + \delta = 1)\end{aligned} \tag{3}$$

In the above equation, p is an input bug localization instance, $\text{CLA}_{\text{BEC}}(p)$ is the prediction score for p based on all features across the four categories (i.e., score, text, topic model, and metadata features), $\text{CLA}_{\text{BEC,Score}}(p)$, $\text{CLA}_{\text{BEC,Text}}(p)$, $\text{CLA}_{\text{BEC,Topic}}(p)$, and $\text{CLA}_{\text{BEC,Meta}}(p)$ are the prediction scores output by the four category-specific BEC prediction models.

Given an instance p, a category-specific BEC model predicts the effectiveness of the bug localization instance p by applying their simple classifiers on p. The prediction score of p is computed by taking the difference between the number of simple classifiers that predict p as effective and those that predict p as ineffective. If the prediction score of p is greater than zero, the category-specific BEC model predicts it as an effective instance. Otherwise, p is predicted as ineffective. Intuitively, the prediction score of p reflects the trend among the simple classifiers when predicting the effectiveness of p. If there are more simple classifiers that predict p as "effective" than "ineffective", the prediction score of p has a positive value. Similarly, p has a negative prediction score if there are less simple classifiers that predict p as "effective" than "ineffective". Algorithm 2 describes how a category-specific BEC model computes a prediction score for a bug localization instance p.

According to (3), each of the category-specific BEC model has a weight in the range of [0, 1] and the sum of the weights equals to 1. Each weight reflects the influence of the corresponding category-specific BEC model to the final model. We tune the values of the weights α, β, γ, and δ in such a way that they maximize the effectiveness of CLA_{BEC} on the training bug localization instances. To achieve that goal, we perform a grid search process by considering all possible combinations of α, β, γ, and δ in the range of [0, 1] where $\alpha + \beta + \gamma + \delta = 1$. We choose values of α, β, γ, and δ that result in the best F-measure on the training data.

In the deployment phase, we employ the learned final prediction model to predict the effectiveness of bug localization instances whose effectiveness labels are unknown. Given

an input localization instance p, based on the category-specific models learned in the training phase, we first compute the following scores, $\text{CLA}_{\text{BEC,Score}}(p)$, $\text{CLA}_{\text{BEC,Text}}(p)$, $\text{CLA}_{\text{BEC,Topic}}(p)$, and $\text{CLA}_{\text{BEC,Meta}}(p)$, using Algorithm 2. Then, we compute the final prediction score of p using (3) and the tuned values of α, β, γ, and δ learned in the training phase. If the prediction score is greater than zero, p is predicted as effective. Otherwise, we predict p as an ineffective instance.

In APRILE^{+}, we use two underlying/base classification algorithms, i.e., RBF netowrk and SVM, to build two BEC classifiers, i.e., RBF_{BEC} and SVM_{BEC}. By default, we set $NC = 10$ and $SR = 80$ % for RBF_{BEC}, and $NC = 50$ and $SR = 80$ % for SVM_{BEC}.

4 Experimental Evaluation

In this section, we first describe our dataset and experiment settings. Next, we describe the research questions and our experiment results that answer these questions. We finally describe the threats to validity.

4.1 Dataset and Experiment Settings

Dataset We conduct experiments using three sets of bug reports and source code files from AspectJ[3] , Eclipse[4] , and SWT[5] which contain a total of more than three thousands bug reports. The details of our dataset are shown in Table 5. Originally, the dataset is introduced by Zhou et al. to evaluate BugLocator. Later, (Saha et al. 2013) and (Wang and Lo 2014) also utilize these bug reports to evaluate BLUiR and AmaLgam. According to (Zhou et al. 2012), for each software project, *all* bug reports of fixed bugs in the study periods shown in Table 5 are collected. To find the ground truth (i.e., faulty files), BugLocator's authors adopt heuristics proposed by (Bachmann and Bernstein 2009) to link bug reports to source code files. Furthermore, we extend our original dataset by manually collecting issue reports and source code files from Apache Tomcat[6]. We exclude bug reports for which names of faulty files are explicitly mentioned in the summaries and descriptions of the bug reports. For these bugs, it is unnecessary to run bug localization tools (Kochhar et al. 2014). Therefore, it is also unnecessary to use our approach to predict the effectiveness of bug localization instances of these bugs. Details of the dataset are shown in Table 5. In total, we investigate a dataset of 3,800 bug reports. The textual bug reports and ground-truth (i.e., faulty files) of our dataset are publicly available[7].

Effectiveness Criterion We consider a bug localization instance effective if a buggy file can be found in the top-N position in the ranked list. In our experiments, the default value of N is 10.

[3] https://www.st.cs.uni-saarland.de/ibugs/

[4] http://goo.gl/Ojqrrp

[5] https://bugcenter.googlecode.com/files/swt-3.1.zip

[6] http://svn.apache.org/repos/asf/tomcat/trunk/

[7] https://github.com/lebuitienduy/aprile_plus

Table 5 Dataset Summary: Third column (#Bugs) is the number of bug reports. Last column (#Files) is the number of source code files

Project	Study Period	#Bugs	#Files
AspectJ	Jul 2002 - Oct 2006	286	6,485
Eclipse (3.1)	Oct 2004 - Mar 2011	3,075	12,863
SWT (3.1)	Oct 2004 - Apr 2011	98	6,485
Tomcat	Jan 2010 - Jan 2014	341	1,879

Evaluation Metrics We use precision, recall, and F-measure (Han and Kamber 2006) to evaluate the performance of our proposed approach. We use four statistics to calculate precision and recall. These statistics are true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The following are their definitions:

- TP: Number of *effective* bug localization instances that are predicted *correctly*.
- FP: Number of *ineffective* bug localization instances that are predicted *incorrectly*.
- TN: Number of *ineffective* bug localization instances that are predicted *correctly*.
- FN: Number of *effective* bug localization instances that are predicted *incorrectly*

Using the above statistics, we compute precision, recall, and F-measure as follows:

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

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

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

In the above equations, precision is the proportion of true positives among the IR-based bug localization instances that *are predicted as effective*. Recall is the proportion of true positives among the IR-based bug localization instances that *are effective*. Both precision and recall reflect the performance of a prediction model. Usually, there is an inverse relationship between precision and recall where higher precision might result in lower recall (and vice versa). Hence, F-measure, which is the harmonic mean of precision and recall, is usually used as a summary measure which informs whether a gain in precision (recall) outweighs a decrease in recall (precision).

Cross-Validation We perform a ten-fold cross-validation to evaluate the performance of our approach. Cross-validation is a standard method to assess accuracy of prediction models (Han and Kamber 2006). It evaluates whether the result of a prediction model generalizes to an independent test dataset. In ten-fold cross-validation, for each project, we randomly partition its bug reports into 10 distinct subgroups of data. Subsequently, we learn a prediction model on nine subgroups, and test the prediction model on the remaining subgroup. We repeat the process 10 times by using each of the 10 subgroups as test data. Finally, we aggregate all outputs from the ten repetitions, and calculate the final precision, recall, and F-measure.

Text Preprocessing In our experiments, we apply Java's regular expression API to split identifiers to smaller words following the Camel case convention. At first, we construct a regular expression to split a string as follows: "`(?<!(^|[A-Z]))(?=[A-Z])|(?<!^)(?=[A-Z][a-z])`". This regular expression makes sure a string is split according to Camel casing convention. e.g., "processFile"

is split to "process" and "File". It also correctly detects acronyms, e.g., "ASTParser" is split to "AST" and "Parser". Moreover, we use the Porter Stemming algorithm (Porter 1980), which is a popular stemming algorithm, to reduce words to their root forms.

Latent Dirichlet Allocation We employ Stanford Topic Modeling Toolbox[8] to train topic models. We set the number of topics to 5, 10, and 15. We use these numbers of topics to capture high-level concepts that are shared by many bug reports – the less the number of topics, the more abstract (or high-level) the topics are. For the other parameters of LDA we use the default settings of Stanford Topic Modeling Toolbox: $\alpha = 0.01$, $\beta = 0.01$, and number of iterations = 1000.

4.2 Research Questions

We analyze several research questions (RQs) to evaluate the performance of our proposed approach. These RQs are presented in the following paragraphs.

RQ_1: *How good is the performance of APRILE$^+$ when predicting the effectiveness of an IR-based bug localization tool?*

Answer to this research question will shed light on the utility of APRILE$^+$. To answer this research question, we use APRILE$^+$ to predict the effectiveness of BugLocator (Zhou et al. 2012), BLUiR (Saha et al. 2013), and AmaLgam (Wang and Lo 2014) to locate buggy files. For each bug report in our dataset, we set the effectiveness criterion N = 10 and predict if each ranked list produced by BugLocator, BLUiR, and AmaLgam is effective or not.

RQ_2: *How good is APRILE$^+$ compared to other effectiveness prediction approaches?*

Recently, Le and Lo propose an approach to predict the effectiveness of a spectrum-based bug localization tool (Le and Lo 2013; Le et al. 2014a). A spectrum-based bug localization tool analyzes a set of failed and correct execution traces, and computes suspiciousness scores of program elements (e.g., statements). Le and Lo's approach is the first study that predicts the effectiveness of automated debugging tools. To a certain extent, our proposed solution in predicting effectiveness of IR-based bug localization instances is a downgrade of Le and Lo's approach from spectrum-based bug localization to IR-based bug localization. On the other hand, Le and Lo's approach uses features extracted from program execution traces and suspiciousness scores. In IR-based bug localization setting, there is no execution traces; thus we can only run Le and Lo's approach on features that are extracted from suspiciousness scores of files. We use this approach as a baseline to compare with APRILE$^+$. We denote this baseline as $\text{SVM}_{Ext,Score}$. Furthermore, we also compare APRILE$^+$ against APRILE.

RQ_3: *Which of the proposed features best discriminate effective IR-based bug localization instances from ineffective ones?*

[8] http://nlp.stanford.edu/software/tmt/tmt-0.4/

We investigate which features from our list are helpful in predicting effective IR-based bug localization instances. In machine learning, Fisher score are usually used to estimate how much discriminative features are. The Fisher score of a feature is calculated as follows

$$FS(j) = \frac{\sum_{class=1}^{\#class} (\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class} (\frac{1}{n_{class}-1} \sum_{i=1}^{n_{class}} (x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2)} \quad (4)$$

In the above equation, $FS(j)$ is the Fisher score of the j^{th} feature, n_{class} is the number of data points (i.e., number of bug localization instances) with label $class$ (i.e., effective or ineffective), $\bar{x}_j$ is the average value of the j^{th} feature over all data points, $\bar{x}_j^{(class)}$ is the average value of the j^{th} feature over all data points with label $class$. If a feature has a Fisher score of zero, then that feature does not help to discriminate effective IR-based bug localization instances from ineffective ones. On the other hand, a feature is very discriminative if its Fisher score is much greater than zero. In this research question, we investigate the overall most discriminative features for every software project (i.e., AspectJ, Eclipse, SWT, and Tomcat).

RQ_4: *What is the effect of changing the effectiveness criterion on the performance of APRILE$^+$?*

By default, a bug localization instance is deemed effective if the top-N (N=1) file in the ranked list is buggy. This corresponds to the case when a developer is willing to inspect only the first recommended program file. However, developers might be willing to inspect more files. In this research question, we vary the effectiveness criterion by considering N = 5 and N = 10. For each of these effectiveness criteria, we evaluate the performance of our approach.

RQ_5: *Could bug localization instances of one software project be used to learn a model for predicting effectiveness of instance of another software project?*

In new software projects, the amount of bug localization instances are not always sufficient to formulate good training data. Therefore, we investigate the effectiveness of APRILE$^+$ in cross-project setting in this research question. Assuming we have bug localization instances from P different software projects. We learn APRILE$^+$'s prediction model from instances of $P - 1$ projects. This model is then employed to predict effectiveness of bug localization instances from the other project.

RQ_6: *How good is the performance of APRILE$^+$ if only the most discriminative features are selected to construct prediction models?*

Feature selection techniques are proposed to improve the accuracy of classification models. However, by default, feature selection is not integrated into APRILE$^+$. Thus, in this research question, we deploy features selection to our proposed approach by selecting the top K percent features with highest Fisher scores in each category to construct APRILE$^+$'s prediction models. We refer to APRILE$^+$ deployed with feature selection as FSAPRILE$^+$. We use FSAPRILE$^+$ to predict the effectiveness of BugLocator's instances with $K \in \{40\ \%, 60\ \%, 80\ \%, 90\ \%, 95\ \%\}$, and compare its precision, recall, and F-measure to those of APRILE$^+$.

4.3 Results

4.3.1 RQ_1: Overall Performance

Table 6 shows the statistics of effective and ineffective bug localization instances output by BugLocator, BLUiR, and AmaLgam to localize faults for 3,800 bug reports in our dataset. With effectiveness criterion $N = 10$, there are more effective instances than ineffective instances in most of data and bug localization tools. Considering all three bug localization tools (i.e., BugLocator, BLUiR, and AmaLgam), effective instances of BugLocator, BLUiR, and AmaLgam are 2,385, 2,456, and 2,557 respectively, which account for 62.76 %, 63.63 %, and 67.29 % of the total instances, respectively.

Table 7 shows the precision, recall, and F-measure of APRILE$^+$ when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in AspectJ, Eclipse, SWT, and Tomcat dataset. According to the table, APRILE$^+$ achieves an F-measure of 66 % or higher in predicting effectiveness of instances in AspectJ, Eclipse, SWT, and Tomcat project. Noticeably, APRILE$^+$'s F-measure is up to 91.33 % when predicting effectiveness of BLUiR instances in SWT. For Tomcat, APRILE$^+$ achieves an F-measure of 66.19 %, 77.46 %, and 78.53 %, respectively in predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances. We find F-measures of APRILE$^+$ in Tomcat dataset are smaller compared to AspectJ, Eclipse, and SWT dataset. This is because the proportion of effective instances to ineffective instances in Tomcat is less than the ones in AspectJ, Eclipse, and SWT. For the three bug localization tools (i.e., BugLocator, BLUiR, and AmaLgam), we note that APRILE$^+$ achieves comparable average F-measures of 82.09 %, 84.27 %, and 84.23 %, respectively. Furthermore, the F-measure results are comparable to or better than those achieved by other software analytics studies (Seo and Kim 2012; Valdivia Garcia and Shihab 2014; Shihab et al. 2013; Le and Lo 2013).

Table 8 shows detailed results of APRILE$^+$ when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in AspectJ dataset. APRILE$^+$ achieves an F-measure of 88.20 %, 84.40 % and 87.17 % for BugLocator, BLUiR, and AmaLgam, respectively. For BugLocator, APRILE$^+$ is able to correctly predict 157 out of 172 effective instances, and 87 out of 114 ineffective instances. For BLUiR, our approach can correctly identify 138 out of 157 effective instances, and 97 out of 129 ineffective instances. For AmaLgam, it can correctly identify 180 out of 196 effective instances, and 53 out of 90 ineffective instances. Noticeably, without our tool, developers using BugLocator would inspect all instances (i.e., 286 instances), and they would be useless 39.86 % of the time (i.e., 114 out of 286 instances

Table 6 Number of effective and ineffective instances for BugLocator, BLUiR, and AmaLgam (effectiveness criterion $N = 10$)

Project	BugLocator			BLUiR			AmaLgam		
	(+)	(–)	$\sum$	(+)	(–)	$\sum$	(+)	(–)	$\sum$
AspectJ	172	114	286	157	129	286	196	90	286
Eclipse	1978	1097	3075	2010	1065	3075	2059	1016	3075
SWT	78	20	98	86	12	98	88	10	98
Tomcat	157	184	341	203	138	341	214	127	341
Total	2385	1415	3800	2456	1344	3800	2557	1243	3800

Table 7 Precision, Recall, and F-measure of APRILE$^+$ on BugLocator, BLUiR and AmaLgam instances (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	85.33 %	91.28 %	88.20 %
Eclipse	81.45 %	87.46 %	84.35 %
SWT	90.79 %	88.46 %	89.61 %
Tomcat	52.85 %	88.54 %	66.19 %
Average	77.61 %	88.94 %	82.09 %
BLUiR			
AspectJ	81.18 %	87.90 %	84.40 %
Eclipse	79.92 %	88.31 %	83.90 %
SWT	90.80 %	91.86 %	91.33 %
Tomcat	66.32 %	93.10 %	77.46 %
Average	79.56 %	90.29 %	84.27 %
AmaLgam			
AspectJ	82.95 %	91.84 %	87.17 %
Eclipse	82.13 %	86.84 %	84.42 %
SWT	97.18 %	78.41 %	86.79 %
Tomcat	69.82 %	89.72 %	78.53 %
Average	83.02 %	86.65 %	84.23 %

are ineffective). By using our tool's predictions, they are only useless 14.67 % of the time (i.e., 27 out of 184 instances predicted as effective are actually ineffective). Similarly, developers using BLUiR would find that the instances are useless 45.1 % of the time (i.e., 129 out of 286 instances are ineffective). By using our tool's predictions, they are only useless 18.82 % of the time (i.e., 32 out of 170 instances predicted as effective are actually ineffective). Last but not least, developers using AmaLgam would find that the instances are useless 31.47 % of the time (i.e., 90 out of 286 instances are ineffective). By using our tool's predictions, they are only useless 17.05 % of the time (i.e., 37 out of 217 instances predicted as effective are actually ineffective). Furthermore, averaging across the three tools, the precision, recall, and F-measure of our approach for AspectJ are 83.15 %, 90.34 %, and 86.59 %, respectively.

Table 9 shows detailed results of APRILE$^+$ when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in Eclipse. APRILE$^+$ achieves an F-measure of 84.35 %, 83.90 % and 84.42 % for BugLocator, BLUiR, and AmaLgam, respectively. For BugLocator, APRILE$^+$ is able to correctly predict 1730 out of 1978 effective instances, and 703 out of 1097 ineffective instances. For BLUiR, our approach can correctly identify 1775 out of 2010 effective instances, and 619 out of 1065 ineffective instances. For AmaLgam, it can correctly identify 1788 out of 2059 effective instances, and 627 out of 1016 ineffective instances. Noticeably, without our tool, developers using BugLocator would inspect all instances (i.e., 3075 instances), and they would be useless of 35.67 % of the time (i.e., 1097 out of 3075 instances are ineffective). By using our tool's predictions, they are only useless 18.55 % of the time (i.e., 394 out of 2124 instances predicted as effective are actually ineffective). Similarly, developers using BLUiR would find that the instances are useless 34.63 % of the time (i.e., 1065 out of 3075 instances are ineffective). By using our tool's

Table 8 True positives, false positives, true negatives, false negatives, precision, recall, and F-measure of APRILE^{+} on predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in *AspectJ* project

Run	TP	FP	TN	FN	Precision	Recall	F-measure
BugLocator							
1	16	3	7	2	84.21 %	88.89 %	86.49 %
2	15	3	8	2	83.33 %	88.24 %	85.71 %
3	15	1	8	4	93.75 %	78.95 %	85.71 %
4	14	5	8	1	73.68 %	93.33 %	82.35 %
5	15	1	12	1	93.75 %	93.75 %	93.75 %
6	13	6	7	3	68.42 %	81.25 %	74.29 %
7	21	2	4	2	91.30 %	91.30 %	91.30 %
8	14	2	13	0	87.50 %	100.00 %	93.33 %
9	17	1	11	0	94.44 %	100.00 %	97.14 %
10	17	3	9	0	85.00 %	100.00 %	91.89 %
Overall	157	27	87	15	85.33 %	91.28 %	88.20 %
BLUiR							
1	15	1	10	2	93.75 %	88.24 %	90.91 %
2	15	2	9	2	88.24 %	88.24 %	88.24 %
3	14	3	10	1	82.35 %	93.33 %	87.50 %
4	11	7	7	3	61.11 %	78.57 %	68.75 %
5	12	4	12	1	75.00 %	92.31 %	82.76 %
6	14	1	12	2	93.33 %	87.50 %	90.32 %
7	16	1	9	3	94.12 %	84.21 %	88.89 %
8	13	5	10	1	72.22 %	92.86 %	81.25 %
9	12	6	10	1	66.67 %	92.31 %	77.42 %
10	16	2	8	3	88.89 %	84.21 %	86.49 %
Overall	138	32	97	19	81.18 %	87.90 %	84.40 %
AmaLgam							
1	22	2	4	0	91.67 %	100.00 %	95.65 %
2	19	3	4	2	86.36 %	90.48 %	88.37 %
3	17	2	7	2	89.47 % 1	89.47 %	89.47 %
4	17	7	3	1	70.83 %	94.44 %	80.95 %
5	18	4	5	2	81.82 %	90.00 %	85.71 %
6	19	2	6	2	90.48 %	90.48 %	90.48 %
7	18	3	5	3	85.71 %	85.71 %	85.71 %
8	17	2	7	3	89.47 %	85.00 %	87.18 %
9	15	10	3	1	60.00 %	93.75 %	73.17 %
10	18	2	9	0	90.00 %	100.00 %	94.74 %
Overall	180	37	53	16	82.95 %	91.84 %	87.17 %

Column "Run" represents each run in cross validation, "TP" is the number of true positive cases, "FP" is the number of false positive cases, "TN" is the number of true negative case, and "FN" is the number of false negative cases

predictions, they are only useless 20.08 % of the time (i.e., 446 out of 2221 instances predicted as effective are actually ineffective). Last but not least, developers using AmaLgam

Table 9 True positives, false positives, true negatives, false negatives, precision, recall, and F-measure of APRILE^{+} on predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in *Eclipse* project

Run	TP	FP	TN	FN	Precision	Recall	F-measure
BugLocator							
1	177	45	59	26	79.73 %	87.19 %	83.29 %
2	182	35	69	21	83.87 %	89.66 %	86.67 %
3	173	37	76	21	82.38 %	89.18 %	85.64 %
4	155	38	81	33	80.31 %	82.45 %	81.36 %
5	173	37	77	20	82.38 %	89.64 %	85.86 %
6	170	49	68	21	77.63 %	89.01 %	82.93 %
7	171	40	66	31	81.04 %	84.65 %	82.81 %
8	182	33	71	22	84.65 %	89.22 %	86.87 %
9	179	47	57	25	79.20 %	87.75 %	83.26 %
10	168	33	79	28	83.58 %	85.71 %	84.63 %
Overall	1730	394	703	248	81.45 %	87.46 %	84.35 %
BLUiR							
1	181	41	59	26	81.53 %	87.44 %	84.38 %
2	180	38	63	26	82.57 %	87.38 %	84.91 %
3	179	55	54	19	76.50 %	90.40 %	82.87 %
4	168	48	73	18	77.78 %	90.32 %	83.58 %
5	173	50	61	23	77.58 %	88.27 %	82.58 %
6	176	53	54	25	76.86 %	87.56 %	81.86 %
7	161	48	72	27	77.03 %	85.64 %	81.11 %
8	193	37	58	20	83.91 %	90.61 %	87.13 %
9	189	35	58	26	84.38 %	87.91 %	86.10 %
10	175	41	67	25	81.02 %	87.50 %	84.13 %
Overall	1775	446	619	235	79.92 %	88.31 %	83.90 %
AmaLgam							
1	186	40	55	26	82.30 %	87.74 %	84.93 %
2	179	35	61	32	83.64 %	84.83 %	84.24 %
3	177	48	60	22	78.67 %	88.94 %	83.49 %
4	168	46	70	23	78.50 %	87.96 %	82.96 %
5	169	33	70	35	83.66 %	82.84 %	83.25 %
6	185	43	58	22	81.14 %	89.37 %	85.06 %
7	172	47	66	23	78.54 %	88.21 %	83.09 %
8	187	33	57	31	85.00 %	85.78 %	85.39 %
9	192	23	65	28	89.30 %	87.27 %	88.28 %
10	173	41	65	29	80.84 %	85.64 %	83.17 %
Overall	1788	389	627	271	82.13 %	86.84 %	84.42 %

Column "Run" represents each run in cross validation, "TP" is the number of true positive cases, "FP" is the number of false positive cases, "TN" is the number of true negative case, and "FN" is the number of false negative cases

would find that the instances are useless 33.04 % of the time (i.e., 1016 out of 3075 instances are ineffective). By using our tool's predictions, they are only useless 17.87 % of the time

(i.e., 389 out of 2177 instances predicted as effective are actually ineffective). Furthermore, averaging across the three tools, the precision, recall, and F-measure of our approach for Eclipse are 81.17 %, 87.54 %, and 84.22 %, respectively.

Table 10 True positives, false positives, true negatives, false negatives, precision, recall, and F-measure of APRILE^{+} on predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in *SWT* project

Run	TP	FP	TN	FN	Precision	Recall	F-measure
BugLocator							
1	5	2	1	1	71.43 %	83.33 %	76.92 %
2	7	1	0	1	87.50 %	87.50 %	87.50 %
3	9	0	1	0	100.00 %	100.00 %	100.00 %
4	7	2	0	1	77.78 %	87.50 %	82.35 %
5	8	0	2	0	100.00 %	100.00 %	100.00 %
6	5	0	4	1	100.00 %	83.33 %	90.91 %
7	8	0	1	1	100.00 %	88.89 %	94.12 %
8	8	1	0	1	88.89 %	88.89 %	88.89 %
9	5	1	3	1	83.33 %	83.33 %	83.33 %
10	7	0	1	2	100.00 %	77.78 %	87.50 %
Overall	69	7	13	9	90.79 %	88.46 %	89.61 %
BLUiR							
1	7	1	0	1	87.50 %	87.50 %	87.50 %
2	7	1	0	1	87.50 %	87.50 %	87.50 %
3	9	1	0	0	90.00 %	100.00 %	94.74 %
4	8	1	1	0	88.89 %	100.00 %	94.12 %
5	8	1	1	0	88.89 %	100.00 %	94.12 %
6	5	1	1	3	83.33 %	62.50 %	71.43 %
7	10	0	0	0	100.00 %	100.00 %	100.00 %
8	8	1	0	1	88.89 %	88.89 %	88.89 %
9	9	1	0	0	90.00 %	100.00 %	94.74 %
10	8	0	1	1	100.00 %	88.89 %	94.12 %
Overall	79	8	4	7	90.80 %	91.86 %	91.33 %
AmaLgam							
1	6	0	0	3	100.00 %	66.67 %	80.00 %
2	6	1	0	2	85.71 %	75.00 %	80.00 %
3	8	0	1	1	100.00 %	88.89 %	94.12 %
4	5	0	2	3	100.00 %	62.50 %	76.92 %
5	8	0	1	1	100.00 %	88.89 %	94.12 %
6	7	1	1	1	87.50 %	87.50 %	87.50 %
7	9	0	0	1	100.00 %	90.00 %	94.74 %
8	7	0	1	2	100.00 %	77.78 %	87.50 %
9	5	0	1	4	100.00 %	55.56 %	71.43 %
10	8	0	1	1	100.00 %	88.89 %	94.12 %
Overall	69	2	8	19	97.18 %	78.41 %	86.79 %

Column "Run" represents each run in cross validation, "TP" is the number of true positive cases, "FP" is the number of false positive cases, "TN" is the number of true negative case, and "FN" is the number of false negative cases

Table 10 shows detailed results of APRILE$^+$ when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in SWT. APRILE$^+$ achieves an F-measure of 89.61 %, 91.33 % and 86.79 % for BugLocator, BLUiR, and AmaLgam, respectively. For BugLocator, APRILE$^+$ is able to correctly predict 69 out of 78 effective instances, and 13 out of 20 ineffective instances. For BLUiR, our approach can correctly identify 79 out of 86 effective instances, and 4 out of 12 ineffective instances. For AmaLgam, it can correctly identify 69 out of 88 effective instances, and 8 out of 10 ineffective instances. Noticeably, without our tool, developers using BugLocator would inspect all instances (i.e., 98 instances), and they would be useless of 20.41 % of the time (i.e., 20 out of 98 instances are ineffective). By using our tool's predictions, they are only useless 8.14 % of the time (i.e., 7 out of 86 instances predicted as effective are actually ineffective). Similarly, developers using BLUiR would find that the instances are useless 12.24 % of the time (i.e., 12 out of 98 instances are ineffective). By using our tool's predictions, they are only useless 9.2 % of the time (i.e., 8 out of 87 instances predicted as effective are actually ineffective). Last but not least, developers using AmaLgam would find that the instances are useless 10.20 % of the time (i.e., 10 out of 98 instances are ineffective). By using our tool's predictions, they are only useless 2.82 % of the time (i.e., 2 out of 71 instances predicted as effective are actually ineffective). Furthermore, averaging across the three tools, the precision, recall, and F-measure of our approach for SWT are 63 %, 90.45 %, and 74.06 %, respectively.

Table 11 shows detailed results of APRILE$^+$ when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in Tomcat. APRILE$^+$ achieves an F-measure of 66.19 %, 77.46 %, and 78.53 % for BugLocator, BLUiR, and AmaLgam, respectively. For BugLocator, APRILE$^+$ is able to correctly predict 139 out of 157 effective instances, and 60 out of 184 ineffective instances. For BLUiR, our approach can correctly identify 189 out of 203 effective instances, and 42 out of 138 ineffective instances. For AmaLgam, it can correctly identify 192 out of 214 effective instances, and 44 out of 127 ineffective instances. Noticeably, without our tool, developers using BugLocator would inspect all instances (i.e., 341 instances), and they would be useless of 53.96 % of the time (i.e., 184 out of 341 instances are ineffective). By using our tool's predictions, they are only useless 47.14 % of the time (i.e., 124 out of 263 instances predicted as effective are actually ineffective). Similarly, developers using BLUiR would find that the instances are useless 40.47 % of the time (i.e., 138 out of 341 instances are ineffective). By using our tool's predictions, they are only useless 33.68 % of the time (i.e., 96 out of 285 instances predicted as effective are actually ineffective). Last but not least, developers using AmaLgam would find that the instances are useless 37.24 % of the time (i.e., 127 out of 341 instances are ineffective). By using our tool's predictions, they are only useless 30.18 % of the time (i.e., 83 out of 275 instances predicted as effective are actually ineffective). Furthermore, averaging across the three tools, the precision, recall, and F-measure of our approach for Tomcat are 63 %, 90.45 %, and 74.06 %, respectively.

4.3.2 RQ_2: APRILE$^+$ vs. Baselines

Table 12 shows precision, recall, and F-measure of SVM$_{Ext,Score}$, and APRILE when predicting effectiveness of BugLocator instances with effectiveness criterion $N = 10$. Comparing the F-measures of SVM$_{Ext,Score}$ and those of APRILE$^+$ (shown in Table 7), we can note that APRILE$^+$ outperforms SVM$_{Ext,Score}$ on all datasets by up to 17.43 %. Similarly,

Table 11 True positives, false positives, true negatives, false negatives, precision, recall, and F-measure of APRILE$^+$ on predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances in *Tomcat* project

Run	TP	FP	TN	FN	Precision	Recall	F-measure
BugLocator							
1	12	12	7	3	50.00 %	80.00 %	61.54 %
2	11	11	8	4	50.00 %	73.33 %	59.46 %
3	14	15	4	1	48.28 %	93.33 %	63.64 %
4	15	9	9	1	62.50 %	93.75 %	75.00 %
5	15	12	6	1	55.56 %	93.75 %	69.77 %
6	16	13	5	0	55.17 %	100.00 %	71.11 %
7	13	8	10	3	61.90 %	81.25 %	70.27 %
8	13	13	5	3	50.00 %	81.25 %	61.90 %
9	15	12	6	1	55.56 %	93.75 %	69.77 %
10	15	19	0	1	44.12 %	93.75 %	60.00 %
Overall	139	124	60	18	52.85 %	88.54 %	66.19 %
BLUiR							
1	18	12	1	3	60.00 %	85.71 %	70.59 %
2	18	10	5	1	64.29 %	94.74 %	76.60 %
3	17	13	2	2	56.67 %	89.47 %	69.39 %
4	17	12	4	1	58.62 %	94.44 %	72.34 %
5	18	8	6	2	69.23 %	90.00 %	78.26 %
6	18	9	6	1	66.67 %	94.74 %	78.26 %
7	23	4	6	1	85.19 %	95.83 %	90.20 %
8	16	10	6	2	61.54 %	88.89 %	72.73 %
9	22	7	5	0	75.86 %	100.00 %	86.27 %
10	22	11	1	1	66.67 %	95.65 %	78.57 %
Overall	189	96	42	14	66.32 %	93.10 %	77.46 %
AmaLgam							
1	18	6	7	3	75.00 %	85.71 %	80.00 %
2	18	5	8	3	78.26 %	85.71 %	81.82 %
3	21	9	4	0	70.00 %	100.00 %	82.35 %
4	18	10	3	3	64.29 %	85.71 %	73.47 %
5	21	8	5	0	72.41 %	100.00 %	84.00 %
6	18	11	2	3	62.07 %	85.71 %	72.00 %
7	21	7	5	1	75.00 %	95.45 %	84.00 %
8	17	9	3	5	65.38 %	77.27 %	70.83 %
9	20	10	2	2	66.67 %	90.91 %	76.92 %
10	20	8	5	2	71.43 %	90.91 %	80.00 %
Overall	192	83	44	22	69.82 %	89.72 %	78.53 %

Column "Run" represents each run in cross validation, "TP" is the number of true positive cases, "FP" is the number of false positive cases, "TN" is the number of true negative case, and "FN" is the number of false negative cases

comparing the F-measures of APRILE against that of APRILE$^+$, we can note that APRILE$^+$ outperforms APRILE on all datasets by up to 10.51 %.

Table 12 Precision, recall, and F-measure of $SVM_{Ext,Score}$ and APRILE on predicting effectiveness of BugLocator instances (effectiveness label $N = 10$)

Baseline	Project	Precision	Recall	F-measure
$SVM_{Ext,Score}$	AspectJ	60.14 %	100.00 %	75.11 %
	Eclipse	64.33 %	100.00 %	78.29 %
	SWT	79.59 %	100.00 %	88.64 %
	Tomcat	56.89 %	60.51 %	58.64 %
APRILE	AspectJ	95.35 %	68.62 %	79.81 %
	Eclipse	87.56 %	74.82 %	80.69 %
	SWT	94.87 %	84.09 %	89.16 %
	Tomcat	89.17 %	50.91 %	64.81 %

4.3.3 RQ_3: Most Important Features

Table 13 shows the top-10 features with highest Fisher scores across the four categories of features. According to the table, we find that the score features and text features are the most discriminative ones. In particular, text features are more important for AspectJ and SWT, i.e., top-10 features with the highest Fisher scores for AspectJ and SWT are text features. We believe that there is a correlation between effectiveness labels of bug localization instances and occurrences of words in AspectJ and SWT's textual bug reports, i.e., many effective and ineffective instances are likely described with different sets of words. On the other hand,

Table 13 Overall most important features

Rank	AspectJ	Eclipse
1	provid	Std. Deviation (SS_5)
2	enforc	Variance (SS_4)
3	hope	Mode (SS_3)
4	assum	17^{th} highest score (R_{17})
5	repli	19^{th} highest score (R_{19})
6	suggest	18^{th} highest score (R_{18})
7	singl	20^{th} highest score (R_{20})
8	basic	16^{th} highest score (R_{16})
9	multipl	15^{th} highest score (R_{15})
10	busi	Max. Gap (G_{max})
Rank	SWT	Tomcat
1	effect	Std. Deviation (SS_5)
2	static	Variance (SS_4)
3	correctli	Rel. Diff (RD_{14})
4	shell	Rel. Diff (RD_{13})
5	displai	Rel. Diff (RD_9)
6	public	Rel. Diff (RD_{12})
7	arg	10^{th} highest score (R_{10})
8	sleep	Median (SS_2)
9	void	13^{th} highest score (R_{13})
10	readanddispatch	Rel. Diff (RD_{11})

Table 14 Number of effective and ineffective instances of BugLocator for various effectiveness criteria

Project	N=1		N=5		N=10	
	(+)	(−)	(+)	(−)	(+)	(−)
AspectJ	65	221	139	147	172	114
Eclipse	974	2101	1694	1381	1978	1097
SWT	35	63	68	30	78	20
Tomcat	53	288	120	221	157	184
Total	1127	2673	2021	1779	2385	1415

score features are more discriminative in Eclipse and Tomcat as most of the features in the top-10 are from the suspiciousness score category. This is likely because the differences between the suspiciousness scores of faulty and non-faulty files in Eclipse and Tomcat are significant enough to distinguish effective from ineffective instances. Overall, text features are important for the AspectJ and SWT dataset, and score features are the most important for the Eclipse and Tomcat dataset.

Discussion Table 13 indicates the most important features for each project is considerably different from one another. Noticeably, the overall most discriminative features for AspectJ and SWT are all text features, which are different from important features for Eclipse and AspectJ (see Table 13). The difference is due to the diverse characteristics of issue reports and source code files between Eclipse and Tomcat against the other two projects (i.e., AspectJ and SWT).

Table 15 Precision, Recall, and F-measure of APRILE^{+} for N=1, N=5, and N=10

Project	Precision	Recall	F-measure
N=1			
AspectJ	69.23 %	83.08 %	75.52 %
Eclipse	59.79 %	77.41 %	67.47 %
SWT	71.79 %	80.00 %	75.68 %
Tomcat	25.76 %	64.15 %	36.76 %
Average	56.64 %	76.16 %	63.86 %
N=5			
AspectJ	85.16 %	78.42 %	81.65 %
Eclipse	77.20 %	86.36 %	81.53 %
SWT	84.06 %	85.29 %	84.67 %
Tomcat	51.40 %	76.67 %	61.54 %
Average	74.37 %	81.69 %	77.35 %
N=10			
AspectJ	85.33 %	91.28 %	88.20 %
Eclipse	81.45 %	87.46 %	84.35 %
SWT	90.79 %	88.46 %	89.61 %
Tomcat	52.85 %	88.54 %	66.19 %
Average	77.61 %	88.94 %	82.09 %

4.3.4 RQ_4: Effect of Varying Effectiveness Criterion

Table 14 shows the number of effective and ineffective BugLocator instances for N = 1, N = 5 and N = 10. Clearly, the number of effective instances increases when we vary the value of N from 1 to 10. Overall, the numbers of effective BugLocator instances for N equals to 1, 5, and 10, are 29.66 %, 53.18 %, and 62.76 % of all BugLocator instances, respectively.

Table 15 shows the effect of using various effectiveness criteria on the performance of APRILE$^+$. We notice that the average F-measure increases from 63.86 % to 82.09 % when N varies from 1 to 10. That indicates APRILE$^+$ still performs well in various effectiveness criteria.

4.3.5 RQ_5: Cross-project Setting

To answer this research question, we apply APRILE$^+$ to predict the effectiveness of BugLocator, BLUiR, and AmaLgam instances with effectiveness criteria N = 10 in cross-project setting. Table 16 shows precision, recall, and F-measure of APRILE$^+$ among the investigated bug localization tools. According to the table, APRILE$^+$ achieves the overall F-measure of 47.57 %, 54.33 %, and 69.02 % when predicting effectiveness of BugLocator, BLUiR, and AmaLgam instances, respectively. Noticeably, compared to Table 7 (i.e., standard cross-validation setting), we find F-measure of APRILE$^+$ significantly reduces in most of projects and bug localization tools. This is as expected since cross-project prediction is much harder than within-project prediction – c.f., (Zimmermann et al. 2009). However, F-measures of APRILE$^+$ remain high (up to 90.91 %) in SWT for all instances of BugLocator, BLUiR, and AmaLgam. This implies that it is still possibly to use bug localization

Table 16 Precision, Recall, and F-measure of APRILE$^+$ in cross-project setting (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	73.68 %	8.14 %	14.66 %
Eclipse	81.52 %	33.67 %	47.66 %
SWT	82.76 %	92.31 %	87.27 %
Tomcat	49.63 %	42.68 %	45.89 %
Overall	77.41 %	34.34 %	47.57 %
BLUiR			
AspectJ	61.76 %	13.38 %	21.99 %
Eclipse	79.02 %	39.15 %	52.36 %
SWT	88.89 %	93.02 %	90.91 %
Tomcat	58.43 %	76.85 %	66.38 %
Overall	75.27 %	42.51 %	54.33 %
AmaLgam			
AspectJ	78.18 %	21.94 %	34.26 %
Eclipse	71.49 %	68.58 %	70.00 %
SWT	95.83 %	78.41 %	86.25 %
Tomcat	65.2 %	83.18 %	73.10 %
Overall	71.66 %	66.56 %	69.02 %

instances of one or many projects to infer APRILE$^+$'s models for predicting effectiveness of instances in another project.

4.3.6 RQ_6: APRILE$^+$ with Feature Selection

Table 17, 18, 19, 20, 21 show the average precision, recall, and F-measure of FSAPRILE$^+$ in various settings. According to the table, FSAPRILE$^+$ achieves average F-measures of 77 % or higher. Compared to the effectiveness of APRILE$^+$ shown in Table 7, the F-measure of FSAPRILE$^+$ is lower (for $K \in \{40\ \%, 60\ \%, 80\ \%\}$), but remains comparable to or better than those achieved by other software analytics studies (Le and Lo, 2013; Seo and Kim, 2012; Shihab et al. 2013; Valdivia Garcia and Shihab, 2014). For $K \in \{90\ \%, 95\ \%\}$, the F-measure of FSAPRILE$^+$ is comparable to that of APRILE$^+$. Therefore, we conclude that most of the features contribute to the performance of APRILE$^+$; removing a small percentage of features do not affect the performance much, while the performance is adversely affected when a substantial number of features are removed (e.g., more than 20 %).

5 Discussions

5.1 Why Bagging is Good?

Bagging (i.e., bootstrap aggregation) is an ensemble learning method that can potentially reduce the variance of a learned classification model (aka. a classifier). Model variance corresponds to error due to sensitivity of the model to small fluctuations in a training data.

Table 17 Precision, Recall, and F-measure of FSAPRILE$^+$ on BugLocator, BLUiR, and AmaLgam instances for K=40 % (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	89.15 %	66.86 %	76.41 %
Eclipse	81.93 %	83.67 %	82.79 %
SWT	90.28 %	83.33 %	86.67 %
Tomcat	52.90 %	87.26 %	65.87 %
Average	78.56 %	80.28 %	77.94 %
BLUiR			
AspectJ	84.56 %	73.25 %	78.50 %
Eclipse	79.60 %	80.75 %	80.17 %
SWT	96.23 %	59.30 %	73.38 %
Tomcat	65.98 %	94.58 %	77.73 %
Average	81.59 %	76.97 %	77.45 %
AmaLgam			
AspectJ	84.86 %	80.10 %	82.41 %
Eclipse	82.02 %	83.73 %	82.86 %
SWT	96.97 %	72.73 %	83.12 %
Tomcat	67.89 %	94.86 %	79.14 %
Average	82.94 %	82.86 %	81.88 %

Table 18 Precision, Recall, and F-measure of FSAPRILE$^+$ on BugLocator, BLUiR, and AmaLgam instances for K=60 % (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	88.89 %	65.12 %	75.17 %
Eclipse	83.20 %	80.13 %	81.64 %
SWT	98.21 %	70.51 %	82.09 %
Tomcat	55.97 %	86.62 %	68.00 %
Average	81.57 %	75.59 %	76.72 %
BLUiR			
AspectJ	86.51 %	69.43 %	77.03 %
Eclipse	83.39 %	82.94 %	83.16 %
SWT	95.83 %	53.49 %	68.66 %
Tomcat	65.74 %	93.60 %	77.24 %
Average	82.87 %	74.87 %	76.52 %
AmaLgam			
AspectJ	87.63 %	83.16 %	85.34 %
Eclipse	84.26 %	82.42 %	83.33 %
SWT	95.59 %	73.86 %	83.33 %
Tomcat	69.42 %	94.39 %	80.00 %
Average	84.23 %	83.46 %	83.00 %

High variance potentially causes overfitting. An overfit classifier usually has poor predictive performance on unseen dataset as it is adversely affected by random errors or noise.

Our problem of predicting the effectiveness of bug localization instances is prone to noise, which is likely to translate to higher model variance. There are many types of bugs, each with their own peculiarities. Moreover, many program elements returned by a bug localization technique may share the same scores, and thus the top-N most suspicious program elements are decided by arbitrarily breaking ties.

We hypothesize that bagging is good for our approach since it reduces model variance. To validate this hypothesis, we conduct an empirical analysis using the bug localization instance dataset shown in Table 5. We use the *bias-variance decomposition method* (Kohavi et al. 1996) to infer the variance of classifiers built using SVM and bagging-based SVM (i.e., $\text{SVM}_{\text{bagging}}$). We build different classifiers with different features (i.e., score, text, topic model, and metadata features) extracted from different datasets. Table 22 shows the variance of SVM and $\text{SVM}_{\text{bagging}}$ classifiers. From the table, we note that for 9 out of the 16 dataset-feature type pairs, SVM models have higher variance than $\text{SVM}_{\text{bagging}}$ models. This validates our hypothesis.

5.2 Threats to Validity

Threats to internal validity relate to experimenter errors. We have carefully rechecked our implementation several times, but there may still be errors that we do not notice. Thus, we make APRILE$^+$'s data and implementation publicly available for inspection by the public. The other threat to external validity is our choice of classification algorithms to construct prediction models. We selected only two algorithms: Support Vector Machine and Radial Basis Function Network. There are still other potential algorithms that are more accurate

Table 19 Precision, Recall, and F-measure of FSAPRILE$^+$ on BugLocator, BLUiR, and AmaLgam instances for K=80 % (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	90.21 %	75.00 %	81.90 %
Eclipse	84.85 %	78.16 %	81.37 %
SWT	96.72 %	75.64 %	84.89 %
Tomcat	54.89 %	92.99 %	69.03 %
Average	81.67 %	80.45 %	79.30 %
BLUiR			
AspectJ	80.37 %	83.44 %	81.88 %
Eclipse	84.89 %	79.10 %	81.90 %
SWT	89.55 %	69.77 %	78.43 %
Tomcat	65.53 %	94.58 %	77.42 %
Average	80.09 %	81.72 %	79.91 %
AmaLgam			
AspectJ	85.16 %	79.08 %	82.01 %
Eclipse	86.51 %	80.33 %	83.30 %
SWT	94.37 %	76.14 %	84.28 %
Tomcat	68.68 %	90.19 %	77.98 %
Average	83.68 %	81.44 %	81.89 %

than these two algorithm. In the future, we plan to integrate more classification algorithms in APRILE$^+$. LDA is quite sensitive to the calibration of its parameters (i.e., α, β, number of iterations, and number of topics). In this paper, we set the number of topics as 5, 10, and 15. We also fixed the α, β, and number of iterations to the default values of Stanford Topic Modeling Toolbox v0.4.0 implementation of LDA (i.e., $\alpha = 0.01$, $\beta = 0.01$, and number of Gibbs' iterations = 1000). These values may not be optimal, and the result of APRILE and APRILE$^+$ may improve when these parameters are optimized. Threats to external validity relate to the generalizability of our findings. We have only experimented on bug reports from 4 open source projects. Moreover, the projects are all written in Java and use the same bug reporting system (i.e., Bugzilla). In the future, we plan to reduce these threats by experimenting on more projects written in various programming languages and which use various bug reporting systems. We also plan to extend our study to closed source software systems. We have only investigated 3 IR-based bug localization tools. These tools are the latest IR-based bug localization tools proposed in the literature. In the future, we also want to investigate other IR-based bug localization tools. Threats to construct validity relate to the suitability of our evaluation metrics. We use precision, recall, and F-Measure to evaluate our approach. These metrics are well known (Han and Kamber 2006) and have been used in many software engineering studies (Menzies and Marcus 2008; Antoniol et al. 2008; Le and Lo 2013). Thus, we believe there is little threat to construct validity.

6 Related Work

In this section, we highlight related studies on IR-based bug localization and spectrum-based bug (fault) localization and related studies that employ classification techniques to automate software engineering tasks. The survey here is by no means complete.

Table 20 Precision, Recall, and F-measure of FSAPRILE$^+$ on BugLocator, BLUiR, and AmaLgam instances for K=90 % (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	89.53 %	89.53 %	89.53 %
Eclipse	80.80 %	89.79 %	85.06 %
SWT	89.74 %	89.74 %	89.74 %
Tomcat	51.45 %	90.45 %	65.59 %
Average	77.88 %	89.88 %	82.48 %
BLUiR			
AspectJ	78.74 %	87.26 %	82.78 %
Eclipse	80.16 %	89.65 %	84.64 %
SWT	88.51 %	89.53 %	89.02 %
Tomcat	65.61 %	92.12 %	76.64 %
Average	78.25 %	89.64 %	83.27 %
AmaLgam			
AspectJ	85.19 %	93.88 %	89.32 %
Eclipse	83.63 %	87.86 %	85.69 %
SWT	91.01 %	92.05 %	91.53 %
Tomcat	67.59 %	91.59 %	77.78 %
Average	81.85 %	91.34 %	86.08 %

6.1 IR-Based Bug Localization

These techniques leverage information retrieval techniques to measure the similarity between a bug report and source code files to produce a ranked list of most similar files. Rao and Kak applied many standard IR techniques for bug localization and evaluated their performances (Rao and Kak 2011). Lukins et al. proposed the use of Latent Dirichlet Allocation (LDA) for bug localization (Lukins et al. 2010). Marcus and Maletic used Latent Semantic Indexing (LSI) to recover document to source code traceability links (Marcus and Maletic 2003). Sisman and Kak proposed a technique that predicts the likelihood of a file to be buggy by learning from information stored in version history and use these likelihoods along with a Vector Space Model (VSM) to perform bug localization (Sisman and Kak 2012). Zhong et al. proposed BugLocator, a bug localization tool that uses a specialized VSM model (Zhou et al. 2012). Saha et al. used the structure of source code files and bug reports to build a structured retrieval model for bug localization (Saha et al. 2013). Tantithamthavorn et al. consider co-change histories to improve performance of BugLocator (Tantithamthavorn et al. 2013). Thomas et al. analyze the impact of input parameters on performance of various IR-based bug localization tools, and introduce a framework for combining results of different bug localization tools (Thomas et al. 2013). Wang and Lo combined the approaches by Sisman and Kak, Zhou et al., and Saha et al. to build a better bug localization tool (Wang and Lo 2014). Le et al. introduced the concept of suspicious words for bug localization and proposed a multi-modal approach that utilizes information from textual bug reports and program execution traces to localize bugs (Le et al. 2015).

In this work, we extend these studies by building an approach that can predict if a ranked list that is output by a bug localization technique is likely to be effective or not. If it is

Table 21 Precision, Recall, and F-measure of FSAPRILE$^+$ on BugLocator, BLUiR, and AmaLgam instances for K=95 % (effectiveness criterion $N = 10$)

Project	Precision	Recall	F-measure
BugLocator			
AspectJ	87.22 %	91.28 %	89.20 %
Eclipse	80.07 %	86.55 %	83.19 %
SWT	90.12 %	93.59 %	91.82 %
Tomcat	52.40 %	90.45 %	66.36 %
Average	77.45 %	90.47 %	82.64 %
BLUiR			
AspectJ	77.51 %	83.44 %	80.37 %
Eclipse	79.80 %	87.26 %	83.37 %
SWT	88.64 %	90.70 %	89.66 %
Tomcat	65.84 %	91.13 %	76.45 %
Average	77.95 %	88.13 %	82.46 %
AmaLgam			
AspectJ	86.45 %	94.39 %	90.24 %
Eclipse	81.24 %	86.01 %	83.56 %
SWT	90.91 %	90.91 %	90.91 %
Tomcat	68.44 %	90.19 %	77.82 %
Average	81.76 %	90.38 %	85.63 %

likely to be ineffective, developers can ignore the ranked list and use conventional debugging methods. Following an ineffective bug localization output wastes developers' time and effort.

6.2 Spectrum-Based Bug (Fault) Localization

These techniques analyze program execution traces and pass/fail outputs of test cases to rank program elements (e.g., program statements). Jones and Harold proposed a suspiciousness formula called Tarantula that assigns suspiciousness scores to program elements (Jones and Harrold 2005). Intuitively, Tarantula considers program elements that appear more in failed executions as more likely to be buggy. Similarly, Abreu et al. proposed a suspiciousness formula called Ochiai (Abreu et al. 2009). Studies have shown that Ochiai outperforms Tarantula in ranking buggy program elements. Borrowing concepts from the data mining community, Lucia et al. investigated the effectiveness of 40 association measures and found that Klosgen and Information Gain are promising measures for bug localization (Lucia et al. 2014). Recently, Xie et al. (2013a) theoretically proved that several families of suspiciousness formulas outperform the others. Xie et al. (2013b) further analyzed the effectiveness of SBFL formulas created by applying a genetic programming algorithm (Yoo 2012). Xuan and Monperrus employed a learning-to-rank algorithm to combine suspiciousness scores computed by different SBFL formulas (Xuan and Monperrus 2014). Le et al. proposed a learning-to-rank solution for spectrum-based fault localization by utilizing likely invariants (Le et al. 2016).

Other studies also analyze execution traces to find buggy program elements. However, these studies do not compute suspiciousness scores. Zeller and Hildebrandt proposed Delta Debugging that search for failure inducing inputs (Zeller and Hildebrandt 2002). Employing

Table 22 Variance of SVM and bagging based SVM (i.e., $SVM_{bagging}$) models on score, text, topic model, and metadata features of BugLocator instances (effectiveness criteria $N = 10$)

Feature Type	SVM	$SVM_{bagging}$	Decrease Percentage
AspectJ			
Score	0.0	0.0	0.0 %
Text	0.0389	0.0003	99.23 %
Topic Model	0.0	0.0	0.0 %
Metadata	0.0257	0.0003	98.83 %
Eclipse			
Score	0.0	0.0	0.0 %
Text	0.008	0.0019	76.25 %
Topic Model	0.0	0.0	0.0 %
Metadata	0.0395	0.0004	98.99 %
SWT			
Score	0.0	0.0	0.0 %
Text	0.0	0.0	0.0 %
Topic Model	0.0	0.0	0.0 %
Metadata	0.0049	0.0	100.0 %
Tomcat			
Score	0.0173	0.0	100.0 %
Text	0.0382	0.0208	45.55 %
Topic Model	0.0185	0.0	100.0 %
Metadata	0.0461	0.0167	63.77 %

this technique, Zeller searched for minimal state difference between failed and successful execution traces (Zeller 2002). Cleve and Zeller extended Delta Debugging further by incorporating cause transitions, which is implemented in a tool named AskIgor (Cleve and Zeller 2005).

In this work, we focus on predicting the effectiveness of an IR-based bug localization tool instead of a spectrum-based bug localization tool. Le and Lo have developed an approach that predicts the effectiveness of a spectrum-based bug localization tool (Le and Lo 2013; Le et al. 2014a). We have used an adapted version of Le and Lo's approach (i.e., SVM^{Ext}_{Score}) as a baseline and demonstrated that our approach can outperform this baseline on all datasets by up to a 94.39 % increase in F-measure.

6.3 Classification Techniques for Software Engineering

There are many studies that employ classification techniques to solve software engineering problems. Bowring et al. employed active learning to predict whether an execution trace is correct or faulty (Bowring et al. 2004). Brun and Ernst applied classification algorithms to generate prediction models that classify fault-revealing properties of code (Brun and Ernst 2004). Antoniol et al. proposed an approach that predicts whether an issue is a bug report or a feature request (Antoniol et al. 2008). Lamkafi et al., Menzies and Marcus, and Tian et al. proposed techniques that predict the severity of reported bugs (Menzies and Marcus 2008; Lamkanfi et al. 2010; Lamkanfi et al. 2011; Tian et al. 2012a). Jalbert and Weimer proposed an approach that predicts whether a bug report is a duplicate or not (Jalbert and Weimer 2008). Tian et al. extended Jalbert and Weimer's work using a more effective solution (Tian

et al. 2012b). Shibab et al. proposed an approach that predicts whether a closed bug report would be reopened (Shihab et al. 2010).

Different from the above studies, we propose a new classification framework to solve a different software engineering problem, namely the prediction of the effectiveness of IR-based bug localization instances.

6.4 Query Performance Prediction in Information Retrieval

In information retrieval, query performance prediction approaches are the most relevant to our work in predicting effectiveness of bug localization tools. This line of research focuses on estimating query difficulty, i.e., the quality of search results retrieved for a query from a given collection of documents. There are two main groups of query performance prediction approaches: pre-retrieval and post-retrieval. In general, pre-retrieval prediction approaches estimate the quality of query's results before the retrieval takes place (He and Ounis 2004; Mothe and Tanguy 2005). On the hand, post-retrieval prediction approaches analyzes rankings returned by search engines/retrieval systems to estimate quality of query's results (Vinay et al. 2006; Cronen-Townsend et al. 2002; Shtok et al. 2009; 2010). In comparison with query performance prediction, our work is more relevant to post-retrieval approaches based score distribution analysis (Shtok et al. 2009; 2010).

In (Shtok et al. 2009; 2010), Shtok et al. propose NQC (i.e., Normalized Query Commitment) which is a predictor (i.e., measurement) to estimate the query performance. For a given query q, the NQC value of q is the standard deviation of retrieval scores of all documents normalized by the query likelihood retrieval score of the whole corpus. If NQC of q is greater than the mean retrieval score, q is estimated as "difficult". Otherwise, q is an "easy" query. Different from Shtok et al., we compute several statistics from suspiciousness scores (i.e., retrieval scores) such as gaps between suspiciousness scores, relative differences, etc., in addition to standard deviation. Furthermore, we extract textual and context-specific features from bug reports besides features from suspiciousness scores. Furthermore, we employ state-of-the-art machine learning classification algorithms to predict the effectiveness of bug localization instances instead of comparing a statistic based on standard deviation of retrieval scores as Shtok et al.'s approach. However, different from query performance prediction methods, we predict effectiveness of ranked lists returned by an information retrieval based bug localization tool for a given bug report.

7 Conclusion and Future Work

In this paper, we address the unreliability of IR-based bug localization techniques by proposing APRILE$^+$ which is an automatic approach that can predict the effectiveness of a bug localization instance. We propose a number of features that we extract from an input bug report and a ranked list of suspiciousness scores that are output by a bug localization tool. These features include: suspiciousness score features, text features, topic model features, and metadata features. For each feature category, we use the values of the features extracted from a training data to learn a prediction model. These models are then combined into a composite prediction model in which the relative contributions of the individual models are learned from the training data. We name this solution, which was introduced in our conference paper (Le et al. 2014b), APRILE, which stands for Automated PREdiction of IR-based Bug Localization's Effectiveness. We further integrate APRILE with two other components that are learned using our bagging-based ensemble classification (BEC) method, i.e.,

RBF^{BEC} and SVM^{BEC}. We refer to the extension of APRILE as APRILE$^+$. We evaluate APRILE$^+$ to predict the effectiveness of state-of-the-art bug localization techniques applied on a dataset of 3,800 bugs from AspectJ, Eclipse, and SWT. Our approach can achieve an average precision, recall, and F-measure of 77.61 %, 88.94 %, and 82.09 %, respectively. Furthermore, APRILE$^+$ outperforms a baseline based on the approach proposed by Le and Lo (Le and Lo 2013; Le et al. 2014a) and APRILE by up to a 17.43 % and 10.51 % increase in F-measure, respectively.

For future work, we plan to add more features to improve the F-measure of APRILE$^+$ further. We also plan to evaluate APRILE$^+$ with additional bug reports from various software projects that are implemented in different programming languages to reduce the threats to external validity further. Moreover, we plan to employ other approaches (i.e., learning to rank algorithms or meta-heuristics etc.) to tune coefficients of models in (2) and (3). Furthermore, to reduce the threats of internal validity, we plan to create a robust test suite and share our code using iPython notebook which can be transparently shared.

Dataset and tool release APRILE$^+$'s dataset and source code are publicly available at https://github.com/lebuitienduy/aprile_plus.

References

Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJ (2009) A practical evaluation of spectrum-based fault localization. J Syst Softw 82(11):1780–1792

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: A text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, ACM, New York, NY, USA, CASCON '08, pp 23:304–23:318

Ayewah N, Pugh W (2010) The google findbugs fixit. In: Proceedings of the 19th international symposium on Software testing and analysis, ACM, pp 241–252

Bachmann A, Bernstein A (2009) Software process data quality and characteristics: a historical view on open and closed source projects. In: Proceedings of the joint international and annual ERCIM workshops on principles of software evolution (IWPSE) and software evolution (Evol) workshops, ACM, pp 119–128

Bauer E, Kohavi R (1999) An empirical comparison of voting classification algorithms: bagging, boosting, and variants. Mach Learn 36(1-2):105–139

Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022

Bowring JF, Rehg JM, Harrold MJ (2004) Active learning for automatic classification of software behavior. In: Proc. 2004 int. Symp. on software testing and analysis (ISSTA'04), Boston, MA, pp 195–205

Breiman L (1996a) Bagging predictors. Mach Learn 24(2):123–140. doi:10.1007/BF00058655

Breiman L (1996b) Bagging predictors. Mach Learn 24:123–140

Broomhead DS, Lowe D (1988) Multivariable functional interpolation and adaptive networks. Complex Syst 2:321–355

Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: Proc. 26th Int. Conf Software Engineering (ICSE'04), Edinburgh, Scotland

Cleve H, Zeller A (2005) Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '05, pp 342–351

Cronen-Townsend S, Zhou Y, Croft WB (2002) Predicting query performance. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, pp 299–306

Han J, Kamber M (2006) Data Mining Concepts and Techniques, 2nd edn. Morgan Kaufmann

He B, Ounis I (2004) Inferring query performance using pre-retrieval predictors. In: String processing and information retrieval, Springer, pp 43–54

Heckman S, Williams L (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf Softw Technol 53(4):363–387

Hovemeyer D, Pugh W (2004) Finding bugs is easy. ACM Sigplan Notices 39(12):92–106

Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pp 52–61

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: 2013 35Th international conference on software engineering, ICSE, IEEE, pp 672–681

Johnson S (1978) Lint, a c program checker

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, pp 273–282

Kim S, Ernst MD (2007) Which warnings should i fix first? In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, pp 45–54

Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: do they matter? In: ACM/IEEE International conference on automated software engineering, ASE'14, vasteras, Sweden - september 15 - 19, 2014, pp 803–814

Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, pp 165–176

Kohavi R, Wolpert DH et al (1996) Bias plus variance decomposition for zero-one loss functions. In: ICML, vol 96, pp 275–83

Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: 7Th IEEE working conference on mining software repositories, MSR, IEEE, pp 1–10

Lamkanfi A, Demeyer S, Soetens QD, Verdonck T (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: 15Th european conference on software maintenance and reengineering, CSMR, IEEE, pp 249–258

Le TD, Lo D (2013) Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In: 2013 IEEE International conference on software maintenance, eindhoven, the Netherlands, September 22-28, 2013, pp 310–319

Le TD, Lo D, Thung F (2014a) Should I follow this fault localization tools output? Empirical Software Engineering pp 1–38. doi:10.1007/s10664-014-9349-1

Le TD, Thung F, Lo D (2014b) Predicting effectiveness of ir-based bug localization techniques. In: 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014, pp 335–345

Le TD, Oentaryo RJ, Lo D (2015) Information retrieval and spectrum based bug localization: better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, pp 579–590. doi:10.1145/2786805.2786880

Le TD, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, pp 177–188

Lemmens A, Croux C (2006) Bagging and boosting classification trees to predict churn. J Mark Res 43(2):276–286

Lucia LoD, Jiang L, Thung F, Budi A (2014) Extended comprehensive study of association measures for fault localization. J Softw: Evol Process 26(2):172–219

Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent dirichlet allocation. Inf Softw Technol 52(9):972–990

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press, New York

Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th international conference on software engineering, may 3-10, 2003, Portland, Oregon, USA, pp 125–137

Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: 24Th IEEE international conference on software maintenance (ICSM 2008), september 28 - october 4, 2008, Beijing, China, pp 346–355

Mitchell T (1997) Machine Learning. McGraw Hill

Mothe J, Tanguy L (2005) Linguistic features to predict query difficulty. In: ACM Conference on research and development in information retrieval, SIGIR, Predicting query difficulty-methods and applications workshop, pp 7–10

Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 20th international symposium on software testing and analysis, ISSTA 2011, toronto, ON, Canada, July 17-21, 2011, pp 199–209

Porter MF (1980) An algorithm for suffix stripping. Program 14(3):130–137
Prasad AM, Iverson LR, Liaw A (2006) Newer classification and regression tree techniques: bagging and random forests for ecological prediction. Ecosystems 9(2):181–199
Rao S, Kak AC (2011) Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings, pp 43–52
Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 28Th IEEE/ACM international conference on automated software engineering, ASE 2013, silicon valley, CA, USA, November 11-15, 2013, pp 345–355
Seo H, Kim S (2012) Predicting recurring crash stacks. In: IEEE/ACM International conference on automated software engineering, ASE'12, essen, Germany, September 3-7, 2012, pp 180–189
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto K (2010) Predicting re-opened bugs: A case study on the eclipse project. In: 17Th working conference on reverse engineering, WCRE 2010, 13-16 october, 2010, Beverly, MA, USA, pp 249–258
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, ichi Matsumoto K (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042
Shtok A, Kurland O, Carmel D (2009) Predicting query performance by query-drift estimation. In: Advances in Information Retrieval Theory, Springer, pp 305–312
Shtok A, Kurland O, Carmel D (2010) Using statistical decision theory and relevance models for query-performance prediction. In: Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval, ACM, pp 259–266
Sisman B, Kak AC (2012) Incorporating version histories in information retrieval based bug localization. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, IEEE Press, pp 50–59
Tantithamthavorn C, Ihara A, Matsumoto K (2013) Using co-change histories to improve bug localization performance. In: 14Th ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing, SNPD 2013, honolulu, hawaii, USA, 1-3 July, 2013, pp 543–548
Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology Planning Report 02–32002
Thomas SW, Nagappan M, Blostein D, Hassan AE (2013) The impact of classifier configuration and classifier combination on bug localization. IEEE, Trans Software Eng 39(10):1427–1443
Tian Y, Lo D, Sun C (2012a) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 19Th working conference on reverse engineering, WCRE, IEEE, pp 215–224
Tian Y, Sun C, Lo D (2012b) Improved duplicate bug report identification. In: 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012, pp 385–390
Valdivia Garcia H, Shihab E (2014) Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, pp 72–81
Vinay V, Cox IJ, Milic-Frayling N, Wood K (2006) On ranking the effectiveness of searches. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, pp 398–404
Wang S, Lo D (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, ACM, pp 53–63
Xia X, Bao L, Lo D, Li S (2016) automated debugging considered harmful considered harmful – a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)
Xie X, Chen TY, Kuo FC, Xu B (2013a) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Methodol 22(4):31
Xie X, Kuo FC, Chen TY, Yoo S, Harman M (2013b) Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: International Symposium on Search Based Software Engineering, Springer, pp 224–238
Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE Computer Society, pp 191–200
Yoo S (2012) Evolving human competitive spectra-based fault localisation techniques. In: International Symposium on Search Based Software Engineering, Springer, pp 244–258

Zeller A (2002) Isolating cause-effect chains from computer programs. In: Proceedings of the tenth ACM SIGSOFT symposium on foundations of software engineering 2002, charleston, south carolina, USA, November 18-22, 2002, pp 1–10

Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200

Zhou J, Zhang H, Lo D (2012) 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, june 2-9, 2012, Zurich, Switzerland, pp 14–24

Zimmermann T, Nagappan N, Gall HC, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the european software engineering conference and the ACM SIGSOFT international symposium on foundations of software engineering, 2009, amsterdam, the Netherlands, August 24-28, 2009, pp 91–100

Tien-Duy B. Le is a Ph.D. candidate at School of Information Systems, Singapore Management University advised by Assistant Professor David Lo and Assistant Professor Lingxiao Jiang. He received his B.Eng. in Computer Science from University of Technology - Vietnam National University in Ho Chi Minh City (HCMUT) in 2012. His research interests include software bug localization, specification mining, and application of data mining and machine learning techniques in software engineering.

Ferdian Thung is a Ph.D. candidate at School of Information Systems, Singapore Management University advised by Assistant Professor David Lo and Assistant Professor Lingxiao Jiang. He received his B.Eng. in Informatics Engineering from School of Electrical Engineering and Informatics, Bandung Institute of Technology in 2011. He then worked as a research engineer for more than a year in School of Information Systems, Singapore Management University. His research interests are in software engineering and data mining area. He has been working on automated prediction techniques, recommendation systems, and empirical study in software engineering.

David Lo is an Associate Professor in School of Information Systems, Singapore Management University. He received his PhD from School of Computing, National University of Singapore in 2008. Before that, he was studying at School of Computer Engineering, Nanyang Technological University and graduated with a B.Eng (Hons I) in 2004. He has published more than 50 publications in reputable software engineering and data mining venues including: ICSE, FSE, ASE, ISSRE, ISSTA, ICSM, KDD, VLDB, ICDE, CIKM, ACL, etc. He has served as a program committee member and/or organizing committee member for various conferences including KDD, ASE, WCRE, MSR, ICSE, ICSM, SocInfo, etc. He has also served as reviewers of various journals including IEEE Transactions on Reliability, TSE, TOSEM, VLDBJ, TKDE, etc.