Query Quality Prediction for Text Retrieval-based Bug Localization

Wenjie Liu, Weiqin Zou*, Bingting Chen, Biyu Cai, Jingxuan Zhang
Nanjing University of Aeronautics and Astronautics, Nanjing, China
{wenwenmu, weiqin, btchen, caibiyu, jxzhang}@nuaa.edu.cn

Abstract—With the aim to help developers better localize bugs, Researchers propose a series of text retrieval bug localization (TRBL) techniques. Such techniques take bug localization as an information retrieval task with a bug report being a query, all code elements being the document corpus, and the retrieved recommended documents being potential buggy code elements. Like any textual retrieval-based recommendation system, the success of TRBL techniques also largely depends on the quality of queries, i.e., bug reports. Knowing in advance whether a query would lead to relevant results (buggy code) is important for developers so that they can for example decide whether they should reformulate the query before wasting limited resources in checking irrelevant results. To this end, we propose an automatic query quality prediction approach for text retrieval-based bug localization. We take it as a typical classification task, by first collecting six categories of features evolving different aspects of bug reports and code, and then applying classical machine learning algorithms to build models to predict whether a bug report query would retrieve relevant buggy code. Through experiments on six projects, our approach could obtain an average accuracy of 72-91%, and F1 score of 71-91% over different TRBL techniques, and outperform existing techniques on average by 6-10.6% in accuracy, and 5.3-11.1% in F1 scores. We further explore the importance of different feature subsets and find several common features that contribute most to prediction performance.

Keywords–Text Retrieval-based Bug Localization, Quality Prediction

I. Introduction

With the increasing scale of software and the change in the way of receiving bugs (e.g., many software systems use bug tracking systems to openly receive bug reports from end users[1]), developers often face the challenge of fixing many defects within a limited time. To help developers fix bugs in a more effective and efficient way, the academic community proposed a series of bug localization techniques to automatically identify the suspicious code for a given bug[2, 3, 4].

*Weiqin Zou is the corresponding author.

Those bug localization techniques can be broadly divided into two types, i.e., dynamic bug localization and static bug localization, depending on whether they need to execute test cases. Dynamic bug localization generally needs to run test cases to collect execution information of code elements under passing and failing tests, and then output a list of code elements with the largest suspicious scores which are calculated based on the collected execution information [39, 40]. Unlike dynamic approaches, static bug localization is mainly to analyze the static information within code and bug reports, by calculating the typically textual similarity (e.g., cosine similarity) between each code entity and a given bug report, and recommending code entities with the largest similarity scores, to developers for their check (the more similar, the more likely that the code entity is relevant for the bug fixing). The relatively much lower computation cost and competitive performance of static bug localization attracted much attention from the academic community [1][2][5]. A series of static bug localization techniques appeared one by one. Textual retrieval-based bug localization (TRBL) is a mainstream category that has received wide attention.

TRBL treats bug localization as a text retrieval task, where a bug report is treated as a query, code files represent the document collection, and the location process equals retrieving relevant code documents from the collection for a given bug report. Like any text retrieval-based system, the success of TRBL techniques heavily depends on the quality of the bug report query itself. If a low-quality query is provided, even a proficient TRBL tool will yield unsatisfactory results. Developers would greatly benefit if they could determine in advance whether a bug report query will produce fruitful retrieval results. This knowledge would enable them to optimize their bug reports, saving time and effort spent on running TRBL tools and inspecting irrelevant code elements. This is particularly helpful if they have very limited resources but have to handle a large number of bug reports.

Towards this end, we propose an automated method to predict bug report query quality for text retrieval-based bug localization. We take it as a classification task, by collecting six categories of features about bug reports and code first, then building a machine learning model and using the model to do query quality prediction. The collected features involve different aspects of bug reports

and code, that embeds both domain-specific knowledge of bug localization as well as general knowledge from the perspective of text retrieval in the natural language area. To understand the generalizability of our technique, we build our models for several typical TRBL techniques, including Blizzard, AmaLgam, BLUiR, BugLocator, and Lucence. And test our models on six open-source projects of different domains and with different code scales. Our major contributions are as follows:

- We build a model based on six categories of features to automatically predict the bug report query quality for text retrieval-based bug localization. The experiments on 5297 bugs show that our approach could obtain an average accuracy and F1 score of 72 $\sim$ 91% and 71 $\sim$ 91% respectively over different TRBL techniques and outperformed the state-of-the-art Q2P [27] by an average of 6 $\sim$ 10.6 percent, and 5.3 $\sim$ 11.1 percent in terms of accuracy and F1 score.
- We investigate the usefulness of different feature groups in predicting query quality for bug localization. We find that models separately built on those feature groups could also achieve promising prediction results for different TRBL techniques, with a performance difference between them being about or up to 2 $\sim$ 3% on average.
- We investigate the importance of features that mostly affect the prediction performance of our models. We find several common features that contribute most to prediction performance over different TRBL techniques.

The structure of the remaining parts is as follows. Section 2 describes the overall framework of our approach. Section 3 describes our experimental setup. Section 4 presents our experimental results. Section 5 discusses the threats to the validity of our study. The last two sections introduce related work and our arrived conclusions.

II. Methodology

1. Overview

We present an approach to predict the query quality of bug reports for text retrieval-based bug localization. Following the definition of query quality in [27], our approach categorizes bug reports into high-quality and low-quality based on their ability to retrieve relevant buggy files from the recommendation list generated by TRBL techniques, without any additional developer-provided information. Fig.1 shows the overall architecture of our approach. We take query quality prediction as a classification task by attempting to apply typical machine learning algorithms to designed features (training phase) and using the obtained model to do such predictions for new bug reports (prediction phase). Details are as follows.

2. Training Phase

In the model training phase, for a software project, we would take a number of bug reports as well as their labels (i.e., high or low-quality) as input, and train a machine learning model as output. More specifically, given a bug report query, we would extract six categories of features for the query and combine its label to form a training instance. Detailed feature extraction and class labeling could be found in the following subsections II-D and II-E. After we get the training dataset, we apply a machine learning algorithm to the dataset to build a prediction model. Theoretically, any binary classification algorithm can be used in our experiments. In this study, we test five typical classifiers and choose the one with the best performance as our classifier. During model training, we also adopt SMOTE [19] strategy to handle the class imbalance problems (the number of instances from two classes is quite different).

3. Prediction Phase

In the prediction phase, we use the model obtained in the training phase to predict the query quality of a new coming bug report for TRBL techniques. Similarly, before prediction, we need to extract six categories of features from the bug report query (just like those training instances). Based on these features, the model would predict the label (i.e., high or low quality) for the bug report. The predicted label would be referred to by the developers who use TRBL techniques to do bug localization. With this label, developers can determine their way of handling bug reports (e.g., maybe reformulate the bug report first before tool running) or retrieved results (e.g., to what extent to accept the results as being true).

4. Feature Extraction

We collect a set of features that embeds both the domain-specific knowledge of bug localization and general knowledge of text retrieval tasks from an NLP perspective. A small subset of the features are directly collected from [27] (published in TOSEM) which also aims to do query quality prediction like us (they can easily be identified by referring to the reference number "[27]" in the following feature description). All features belong to two large categories namely pre-retrieval and post-retrieval, with each category including several subcategories. Pre-retrieval features are those calculated before a query is run while post-retrieval features are those calculated after running the query.

Pre-Retrieval Features. Pre-retrieval features fall into three subcategories, namely *linguistic, statistic, and domain-specific-bug*. Among them, the *statistic* feature group is from [27] while the *linguistic* and *domain-specific-bug* feature groups are newly added by us to help better do query quality prediction.

1) **Linguistic:** Linguistic feature group focuses on searching for ambiguity and polysemy from a query to evaluate its query quality (e.g., ambiguous queries may be related to poor query results), including the following features.

Word length is a proxy metric for word complexity [8]. the word length of a word is calculated as the number

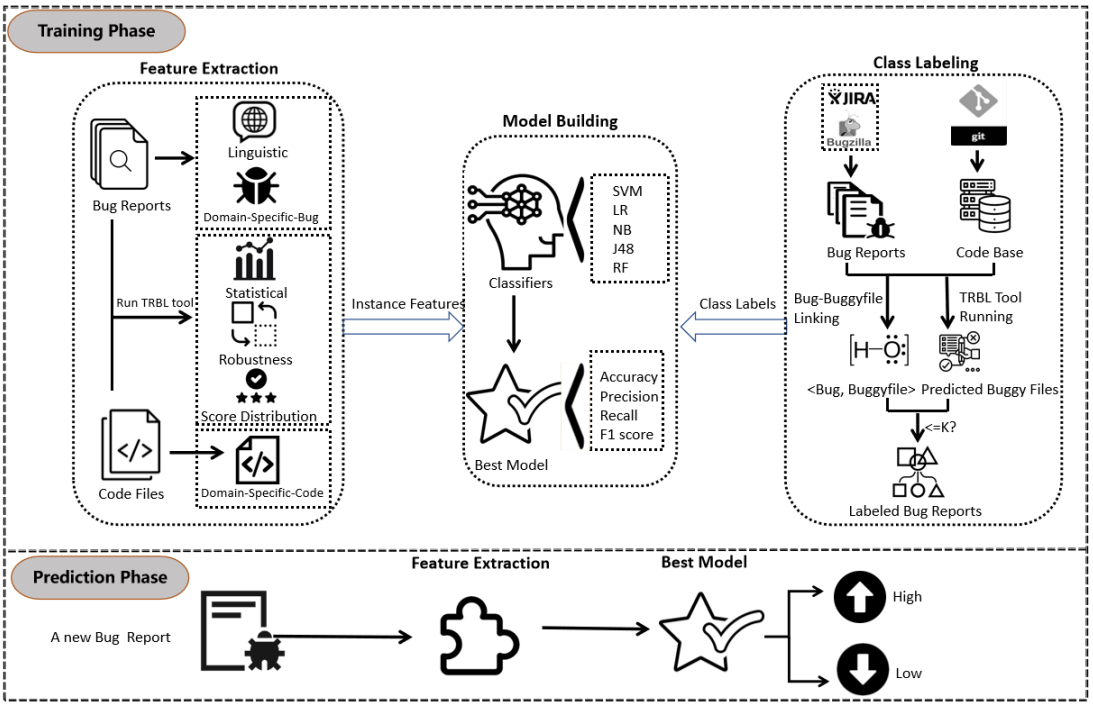


Fig. 1: The overall architecture of our approach

of characters it contains. After we obtain a list of word lengths for words in the bug report, we extract eight features based on these word lengths, namely the maximum, minimum, average, median word length, and the variance, standard deviation of word lengths.

Word numbers is a proxy metric for query complexity [9]. In this study, We use word-num, same-word-num, and diff-word-num as metrics to capture query complexity. Word-num represents the total number of words in a bug report; same-word-num represents the number of words appearing both in a bug report and a code base; while diff-word-num represents the number of words appearing in a bug report but not in a code base.

Irrelevant Syntactical Word Numbers represent the number of conjunctions, prepositions, and pronouns in a bug report query after POS (part of speech) tagging (using NLTK tool respectively. These words are considered rarely captured relevant information[9], hence a good/bad bug report is expected to contain fewer/more conjunctions, prepositions, and pronouns.

Polysemy value involves the definition of polysemy. Polysemy refers to those words having multiple meanings. The more ambiguous a query is, the worse results this query may retrieve [9]. In this paper, we first use wordNet[10] to identify those polysemous words in the bug report query as well as the number of meanings for each polysemy (meanNum). After obtaining these numbers, we further calculate the maximum, minimum, average, median meanNum as well as their standard deviation and variance.

2) **Statistical:** The statistical feature group examines the word distribution in bug reports and the codebase. Nine features(IDF, ICTF, QS, SCS, ENTROPY, SCQ, VAR,CS, PMI) have already been explained in [27], we introduce an additional feature called **Internal Threats.** In data preparation phase, we use some manually summarized heuristic rules to link a bug report to its corresponding buggy files. We have to admit that the rules may be not 100% correct. To avoid the potential bias, we have tried to manually check the linked data carefully and filtered out wrongly linked bug reports. Another threat is that we re-implemented the Q2P approach based on its description from the original paper [27]. We cannot guarantee that we have 100% correctly implemented Q2P for performance comparison. To avoid the potential threats, we conduct several rounds of code review about Q2P.

External Threats. In this study, all experiments and corresponding analysis are conducted on six open source software (OSS) projects programmed in Java. We cannot guarantee that the arrived conclusions or findings could be applicable to other OSS or industry projects written in Java or other languages. However, considering that these projects are well-known and widely-used projects in practice, plus that they come from different domains and are of different sizes, we believe our experiments on these projects still shed some light on the capability of our approach in the real world. ”variability of term occurrences”. This feature concerns the distribution of query terms over the whole collection. In detail, for each query term, we first calculate its tfidf in each document, then

take the standard deviation of these tfidf values as the term occurrence for the term. We extract three features based on the term occurrences of all query terms (noted as TOs), namely the sum-term-occurrence (the sum of the TOs), avg-term-occurrence (the average term occurrence), and the max-term-occurrence (the maximum term occurrence).

3) **Domain-Specific-Bug:** Domain-specific-bug feature group focuses on retrieving bug localization domain specific knowledge related to bug report from the perspective of software practitioners or researchers [13]. Specifically, we collected the following features.

Reporter Experience refers to the experience of bug reporters. Previous studies have found that experienced reporters are better at providing the information needed for bug fixing[14]. In this study, we mainly consider reporters' bug reporting experience and their general working experience. For bug reporting experience, we extract three features, i.e., bug-num, valid-rate, and recent-bug-num.

bug-num represents the total number of bug reports submitted by a reporter. recent-bug-num captures the number of bug reports submitted by a reporter within the last 90 days, considering reporting recency. valid-rate measures the proportion of bug reports successfully fixed over the total bug reports submitted by the reporter. To assess general working experience, we construct a collaboration network based on comment activities within bug reports, similar to previous studies [29, 23]. From this network, we extract nine features, including closeness and total degree, to quantify a reporter's position within the network, as done in [15].

Report Completeness refers to the completeness of technical information (e.g., stack traces and code samples) present in a bug report. A bug report with more complete technical information items is more likely to be located [13, 16, 17]. Inspired by Zimmermann et al. that steps to reproduce, stack traces, code samples, test cases, and screenshots are widely used by developers during their bug fixing [13], we calculate six completeness features based on the appearance or not of these items.

Report Readability refers to the readability of the description content present in the bug report. The readability of the description is found to be an important factor impacting the quality of bug reports [13], which further affects the bug localization and bug fixing [18]. in this paper, we use seven readability measures proposed by previous studies [13], namely flesch, fog, lix, kincaid, ari, coleman-liau, and smog.

Post-Retrieval Properties. Post-retrieval features can be divided into three subcategories: Robustness, Score Distribution, and Domain-Specific-Code. The Score Distribution and five Robustness features are from [27]. To enhance bug localization models, we introduce the Domain-Specific-Code group with five new features from Robustness to improve model performance. Existing features from [27] are not redefined here, and we focus on explaining the newly added features in this study.

4) **Robustness:** Robustness evaluates the stability of the retrieved result list when subjected to perturbations in the query, document list, or retrieval tools [28]. A more stable result indicates a higher-quality and more reliable query. To assess query quality, the difference in search results before and after perturbation is examined. In our study, we consider a total of ten features from the Robustness feature group. As five features (QO, RS, FRC, SAC, CT) have already been explained in [27], we introduce the other five features added in this study as follows.

Query Feedback (qf) Consistency between the original query and the new query generated from its results is measured by the number of common code files (named as common-qf) in their respective top-k code file lists. A higher number of common files indicates greater closeness between the original query and its retrieval results [41]. Additionally, we calculate the differences in maximum, minimum, average, and median retrieval scores (max/min/avg/med-qf) between the two lists, as well as their standard deviation and variance differences (dev/var-qf), to better measure the consistency.

Subpart Overlap (spo) refers to the overlap between the results obtained using the whole query and its subsets. We focus on the overlap of the results by using the report and its two parts: the one-line summary and the problem description. We calculate the overlap separately for each part and then average the results. The overlap is measured by the number of common code files, the differences in maximum, minimum, average, and median retrieval scores (noted as max/min/avg/med-spo) , as well as the standard deviation and variance differences of the retrieval scores (noted as dev/var-spo) between the two lists.

Retrieval Overlap refers to the degree of overlap between the recommendation result lists by different retrieval tools. In our study, we use BugLocator [2] and BRTracer [1] to retrieve results for the same query executed by Lucene in the source code base. We assess overlap based on the number of common code files (common-ro), as well as differences in maximum, minimum, average, median retrieval scores (max/min/avg/med-ro), and the standard deviation and variance of retrieval scores (dev/var-ro).

Codefile Path Span (fps) measures the cohesion between the top-k ranked documents obtained by a query. In software projects where projects are coded according to strict specifications, similar source files are often placed closer together than other files. Therefore we use the fps value between top-k files to measure their cohesion. We use the common part of the absolute paths to calculate the distance between two documents (i.e., source code files). Six feature values based on fps are extracted in the study, namely the maximum/minimum/average/median fps, the standard deviation, and the variance of fps.

Codefile Textual Similarity (fts) measures the textual content cohesion (measured by cosine similarity) between each pair of code files from the top-k file list obtained by a

query. A higher similarity among the top-k files indicates better content cohesion which is related to the quality of a query [28]. In the study, we extract six features based on fts, namely the maximum/minimum/average/median fts as well as their standard deviation and variance.

5) **Domain-Specific-Code:** Domain-specific-code feature group focuses on retrieving bug localization domain-specific knowledge related to code (e.g., Many studies have found that the quality of recommended buggy code file list has an impact on the performance of defect localization [1, 3]. In terms of domain-specific-code, we totally collect eight features from two dimensions namely Codefile Bugfix History and Code Complexity as follows.

Codefile Bugfix History refers to the bug-fixing history of the recommended code file list by TRBL systems. Previous studies have reported the correlation between historical bug fixing and the probability a code file introduces new bugs [2, 7]. These features, including fix-count, fix-developer-count, fix-code-line, and fix-recency. fix-count means the number of bugs whose fixes require modifying the code file. fix-developer-count refers to the number of developers who touched the code file during bug fixing. fix-code-line refers to the number of code lines modified during bug fixing. fix-recency refers to the time distance between the last-fixing-time of a code file and the reporting time of a given bug report.We totally calculated six features based on fix-recency of returned top K files, namely, the maximum/minimum/average/median fix-recency as well as the variance and standard deviation of fix-recency.

Code Complexity is an important metric to measure the quality of code. The more complex a code file is, the more likely it may be to produce bugs [38], hence is more likely to appear in the recommendation list for bug localization [38]. We measure the complexity of a code file by using the number of methods, the number of variables and the number of code lines the code file contains. A more detailed description of all features is available at https://goo.su/kHcnCT.

5. Class Labeling

To create the training dataset, we label each bug report based on its recommended top-K code file list generated by a TRBL tool. As shown in equation (1), following the strategy in [27], High-quality bug reports are those where the top-K list contains truly buggy code files, while low-quality bug reports do not have any truly buggy files in their top-K list. We evaluate our approach using different TRBL techniques on K=20 (the same setting for the state-of-the-art Q2P in [27]).

$$Label = \begin{cases} High, if\ First truthly buggy\ file\ rank <= K \\ Low, otherwise \end{cases} \quad (1)$$

III. Experimental Setup

1. Data Preparation

The goal of data preparation is to create a dataset suitable for training the query quality prediction model. This involves selecting target software products and transforming bug reports from these products into individual instances with features and labels. The whole process includes the following steps.

Target Projets. In this study, we experiment on six open-source projects, namely AspectJ, Tomcat, ZooKeeper, OpenJPA, Hibernate ORM, and Lucene. These projects are mainly written in Java and widely used in TRBL techniques [2, 3]. They vary in size, with line numbers of Java code ranging from 140,175 to 1,762,563 at the time of crawling. Additionally, they come from different domains: ZooKeeper focuses on highly reliable distributed coordination, Hibernate ORM is an Object/Relational Mapping framework, OpenJPA simplifies storing objects in databases, Lucene is an information retrieval software library, AspectJ is an aspect-oriented extension to Java, and Tomcat is an application server and servlet container.

Bug Report–Buggy Files Linking. To construct a benchmark dataset for each selected project, we need to establish a connection between bug reports and the corresponding truly buggy code files. This allows us to determine the class label (high or low quality) for a recommended list of potential buggy files generated by TRBL systems with a list size of K.

For AspectJ and Tomcat projects, Ye et al. share their bug-to-code file links [3]. To avoid duplication of effort, we directly used their linking results. However, some bug reports did not have any deleted or modified code files and were consequently excluded from our datasets. For the remaining four projects, we followed a three-step approach to establish the bug-to-buggy file links.

First, We retrieved 7,609 bug reports with fixed resolutions from the bug tracking systems of the selected projects by the time we crawled (November 2018). Through manual analysis of bug reports and commit logs, we found that developers tended to add projectName-bugID in their commit logs to tell others which bugs they fixed. Hence we use the heuristic rule of projectName-bugID to establish links between bug reports and their corresponding buggy files. For cases where the heuristic rule failed, we further searched their raw bugIDs through commit logs, e.g., using "3217" rather than "ZOOKEEPER-3217". After the two-step search and manual verification, we obtained a total of 6,278 linked bug reports from the four projects.

Following [3], we further removed bug reports that linked to multiple commits or shared the same commit with others, as it was not clear which files were relevant. Some bug reports with no deleted or modified code files were also ignored as it was not applicable to predict buggy files that were not created yet in the buggy version of the project code. Finally, 3,746 bug reports linked to their fixed commits were left. After we linked bug reports with

bug-fixing commits, following the strategy used in [3], for each bug report, we checked out the version right before the bug-fixing commit and took the deleted and modified code files as the buggy files that contained the bug (adding files were ignored as they did not even exist yet when the bug report was initially reported).

Feature Calculating. For each bug report query, we totally extracted six categories of instance features. Section II-D presents the detailed description and calculation for each feature. Among those features, some features (e.g. textual similarity) are calculated over the textual content of bug reports and code files. For these features, we would perform text preprocessing on the bug report and code first before feature calculation. Detailedly, the preprocessing operation is divided into three main steps: text normalization (we removed punctuation and tokenized the content of bug reports and source code files), Stopword/keyword removal (we use the stopword list mentioned in [6] to filter out those insignificant words for bug reports, code files would further remove programming language keywords besides general stopwords), and Stemming (we use the Porter Stemmer to reduce words to their word stem, base, or root form, e.g., predicts to predict).

Experimental DataSets Construction. After we obtain the features for each bug report, we need to label them to get the final experimental dataset for model building. As introduced in the class labeling part (in Section II-E), class labeling of a bug report requires a top-K suspicious code file list generated by a TRBL technique and a truly buggy file set for the bug report (obtained in the Bug Report-Buggy Files Linking part). For developers who use a certain TRBL technique to locate bugs in the codebase, we need to have a corresponding dataset for the TRBL technique, so that we can build a model to predict the query quality of a given bug report for the specific TRBL technique used by developers at hand. By running a TRBL technique over bug reports of a project at K recommendation level, and using the linked bug reports and truly buggy files, we can easily obtain an experimental dataset for the TRBL technique with a certain K (Following [27], we set K=20). We test our models on five classical TRBL techniques, namely Lucene, BugLocator[2], BLUiR [5], Blizzard [4], and AmaLgam [7]. These techniques are strong baselines in TRBL research and are reported to have good performance [3, 6]. Testing over various TRBL techniques could help us have an objective view of the general performance of our models in the TRBL area. After class labeling, for datasets, the numbers of bug reports with high/low-quality for different TRBL techniques are shown in Table I.

Table I: The numbers of bug reports labeled with High/Low-quality for different TRBL techniques

Project		Lucene	BugLocator	Bluir	AmaLgam	Blizzard
Aspectj	high	307	389	341	413	329
	low	256	174	222	150	234
Tomcat	high	769	855	673	860	830
	low	223	137	319	132	162
OpenJPA	high	349	403	429	426	371
	low	184	130	104	107	162
Hibernate ORM	high	803	983	715	1017	871
	low	482	302	570	268	414
Lucene	high	1155	1286	1108	1319	1293
	low	299	168	346	135	161
ZooKeeper	high	405	432	387	436	421
	low	65	38	83	34	49

2. *Classifiers and Evaluation Metrics*

To understand which classifier is best suitable for our task, we build our models by applying different classical machine learning algorithms that are widely used in software engineering research [24, 25], including Naive Bayes, Support Vector Machine, Decision Tree, Logistic Regression, and Random Forest. As a binary classification task, we use four typical metrics to measure the overall performance of our query quality prediction approach for text retrieval-based bug localization, namely Accuracy, Precision, Recall, and F1 Score.

IV. Experiment Results

To understand the effectiveness of our approach in predicting bug report query quality for text retrieval-based bug localization, we investigate the following three questions.

RQ1. Can we effectively predict the query quality of a bug report for bug localization tasks?

In this RQ, we try to build a machine learning classifier based on the features we extract and test its performance on our experimental datasets. As we have no idea which classifier is best suitable for our task, we check five typical machine learning models, including NB, SVM, J48, RF, and LR. 10-fold cross-validation is conducted over the datasets to obtain the final performance of a model in terms of accuracy, precision, recall, and F1 score. Table II shows the results of different machine learning models on the datasets of five TRBL techniques.

Classifier with Best Performance. From Table II, we can find that random forest performs best over all five classifiers in terms of precision, recall, accuracy, and F1 score. Among five classifiers, NB is found to perform worst in most projects under five bug localization techniques. The above results indicate that Random Forest would be a good choice if we attempt to build a query quality prediction model for various TRBL techniques.

> Finding 1.Random Forest Classifier performed best in terms of all performance metrics (e.g., accuracy), when compared to other typical classifiers (NB, SVM, J48 and LR) over different TRBL techniques.

Random Forest Performance. From Table II, we can observe that on the whole, RF can obtain an accuracy (A), precision (P), recall (R), and F1 score (F) as high as 0.983, with the lowest A, P, R, F being 0.698, 0.698, 0.698 and 0.696 respectively, for all TRBL techniques. The average

Table II: Prediction performance of each classifier on six projects against five bug localization techniques

Project		Lucene				BugLocaor				Bluir				AmaLgam				Blizzard			
		A	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**
Aspectj	J48	0.564	0.564	0.564	0.564	0.648	0.651	0.648	0.648	0.614	0.613	0.614	0.613	0.704	0.704	0.704	0.703	0.619	0.614	0.619	0.616
	RF	**0.698**	**0.698**	**0.698**	**0.696**	**0.770**	**0.773**	**0.770**	**0.768**	**0.710**	**0.713**	**0.710**	**0.701**	**0.849**	**0.849**	**0.849**	**0.849**	**0.733**	**0.738**	**0.733**	**0.721**
	NB	0.628	0.666	0.628	0.595	0.531	0.578	0.531	0.487	0.557	0.545	0.557	0.542	0.581	0.605	0.581	0.542	0.599	0.597	0.599	0.598
	SVM	0.676	0.676	0.676	0.675	0.587	0.588	0.587	0.587	0.566	0.551	0.566	0.541	0.650	0.653	0.650	0.646	0.670	0.668	0.670	0.652
	LR	0.624	0.624	0.624	0.624	0.562	0.562	0.562	0.562	0.549	0.545	0.549	0.546	0.642	0.642	0.642	0.642	0.587	0.583	0.587	0.585
Tomcat	J48	0.745	0.745	0.745	0.745	0.794	0.794	0.794	0.794	0.615	0.616	0.615	0.616	0.796	0.796	0.796	0.796	0.776	0.776	0.776	0.776
	RF	**0.868**	**0.868**	**0.868**	**0.868**	**0.937**	**0.939**	**0.937**	**0.937**	**0.760**	**0.761**	**0.760**	**0.759**	**0.946**	**0.946**	**0.946**	**0.946**	**0.924**	**0.926**	**0.924**	**0.924**
	NB	0.686	0.689	0.686	0.686	0.569	0.623	0.569	0.526	0.535	0.544	0.535	0.525	0.596	0.636	0.596	0.554	0.567	0.622	0.567	0.517
	SVM	0.723	0.725	0.723	0.724	0.646	0.647	0.646	0.645	0.554	0.554	0.554	0.554	0.644	0.646	0.644	0.641	0.630	0.633	0.630	0.629
	LR	0.701	0.701	0.701	0.701	0.654	0.654	0.654	0.654	0.550	0.550	0.550	0.549	0.638	0.638	0.638	0.637	0.636	0.636	0.636	0.636
OpenJPA	J48	0.736	0.736	0.736	0.736	0.700	0.696	0.696	0.696	0.765	0.766	0.765	0.765	0.762	0.762	0.762	0.762	0.618	0.618	0.618	0.618
	RF	**0.847**	**0.847**	**0.847**	**0.847**	**0.840**	**0.841**	**0.840**	**0.840**	**0.901**	**0.904**	**0.901**	**0.901**	**0.876**	**0.876**	**0.876**	**0.876**	**0.751**	**0.752**	**0.751**	**0.749**
	NB	0.684	0.686	0.684	0.684	0.638	0.655	0.638	0.629	0.622	0.655	0.622	0.602	0.641	0.657	0.641	0.631	0.565	0.565	0.565	0.565
	SVM	0.790	0.790	0.790	0.790	0.648	0.655	0.648	0.645	0.709	0.716	0.709	0.707	0.680	0.686	0.680	0.677	0.593	0.592	0.593	0.592
	LR	0.728	0.728	0.728	0.728	0.614	0.614	0.614	0.614	0.702	0.702	0.702	0.702	0.653	0.654	0.653	0.653	0.583	0.582	0.583	0.582
Hibernate ORM	J48	0.684	0.683	0.684	0.683	0.704	0.704	0.704	0.704	0.628	0.626	0.628	0.627	0.736	0.737	0.736	0.736	0.638	0.638	0.638	0.638
	RF	**0.806**	**0.806**	**0.806**	**0.805**	**0.843**	**0.851**	**0.843**	**0.842**	**0.707**	**0.710**	**0.707**	**0.683**	**0.882**	**0.886**	**0.882**	**0.882**	**0.756**	**0.759**	**0.756**	**0.755**
	NB	0.636	0.635	0.636	0.626	0.537	0.602	0.537	0.475	0.624	0.506	0.574	0.498	0.575	0.619	0.575	0.523	0.504	0.539	0.504	0.422
	SVM	0.719	0.718	0.719	0.717	0.608	0.608	0.608	0.608	0.639	0.629	0.639	0.579	0.619	0.619	0.619	0.617	0.561	0.560	0.561	0.560
	LR	0.720	0.720	0.720	0.720	0.609	0.608	0.609	0.609	0.596	0.577	0.596	0.58	0.623	0.623	0.623	0.623	0.556	0.555	0.556	0.555
Lucene	J48	0.811	0.810	0.811	0.810	0.843	0.843	0.843	0.843	0.694	0.695	0.694	0.694	0.851	0.851	0.851	0.851	0.833	0.833	0.833	0.833
	RF	**0.907**	**0.907**	**0.907**	**0.907**	**0.954**	**0.955**	**0.954**	**0.954**	**0.856**	**0.862**	**0.856**	**0.855**	**0.969**	**0.971**	**0.969**	**0.969**	**0.964**	**0.965**	**0.964**	**0.964**
	NB	0.661	0.722	0.661	0.645	0.611	0.648	0.611	0.591	0.566	0.591	0.566	0.542	0.590	0.643	0.590	0.556	0.635	0.643	0.635	0.631
	SVM	0.785	0.787	0.785	0.784	0.695	0.697	0.695	0.695	0.603	0.604	0.603	0.603	0.684	0.685	0.684	0.684	0.669	0.672	0.669	0.669
	LR	0.772	0.772	0.772	0.772	0.687	0.687	0.687	0.687	0.593	0.593	0.593	0.593	0.681	0.681	0.681	0.681	0.660	0.660	0.660	0.660
ZooKeeper	J48	0.794	0.796	0.794	0.794	0.850	0.852	0.850	0.850	0.734	0.736	0.734	0.735	0.886	0.887	0.886	0.886	0.825	0.825	0.825	0.825
	RF	**0.946**	**0.946**	**0.946**	**0.946**	**0.976**	**0.976**	**0.976**	**0.976**	**0.893**	**0.893**	**0.893**	**0.893**	**0.983**	**0.983**	**0.983**	**0.983**	**0.967**	**0.967**	**0.967**	**0.967**
	NB	0.766	0.767	0.766	0.766	0.715	0.718	0.715	0.714	0.636	0.662	0.636	0.630	0.771	0.778	0.771	0.769	0.722	0.758	0.772	0.712
	SVM	0.828	0.832	0.828	0.828	0.800	0.807	0.800	0.799	0.652	0.653	0.653	0.653	0.854	0.864	0.854	0.853	0.808	0.814	0.808	0.808
	LR	0.798	0.798	0.798	0.798	0.789	0.789	0.789	0.789	0.651	0.650	0.651	0.650	0.818	0.818	0.818	0.818	0.796	0.796	0.796	0.796

#A represents Accuracy. P represents Precision. R represents Recall. F represents F1-Sore.

A, P, R, and F1 score values are all larger than 0.8, which means our approach is promising in predicting whether a bug report query would lead to a good retrieval result list for various TRBL techniques.

Finding 2.Random Forest Classifiers could achieve an average accuracy of 72.6 $\sim$ 91.8% and the average F1 score of 71.5 $\sim$ 91.8% for the five TRBL techniques over six projects.

Performance Improvement over Q2P. To understand whether and how much our approach outperforms existing approaches, we further compare the performance values of our RF model with Q2P [27] which also conducts query quality prediction. From the Table, we can find that the largest/smallest improvements in accuracy A, precision P, recall R, and F1 score for five TRBL techniques are 11.4/6.0, 11.5/6.6, 11.4/6.0 and 11.5/4.1 percent respectively. And the average improvements of A, P, R, F1 Score are all larger than 8 percent, which indicates our methods are indeed more effective than Q2P.

Finding 3.Our RF models could generally outperform the state-of-the-art Q2P, by achieving an average improvement of 6.0 $\sim$ 10.6 percent in terms of accuracy, and 5.3 $\sim$ 11.1 percent in terms of F1 scores for the five TRBL techniques over six projects.

RQ2. How effective are the prediction models built on all features compared to that built on subset features? In this RQ, we try to understand how individual feature subsets would impact the prediction performance. We conduct two comparisons, one is to compare the performance of pre-retrieval features with post-retrieval features, and the other one is to compare the performance of existing features used by Q2P (noted as Base) with newly added features by us (noted as Our). Doing these comparisons, on one hand, can help developers do better decisions in tool adoption, for example whether to use all features to get the probably best performance or just use some of them to get acceptable performance and on the other hand can validate how useful our newly added features are (which may help lay a better basis for query quality prediction tasks). Like

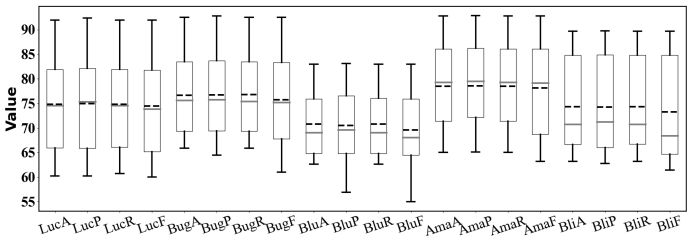
Table III: Performance improvement of our method compared to Q2P in six projects with five techniques

Project	Lucene				BugLocaor				Bluir				AmaLgam				Blizzard			
	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F
Aspectj	9.2	9.0	9.2	9.6	10.0	10.2	10.0	10.0	6.0	6.6	6.0	5.5	10.7	10.5	10.6	10.6	6.0	7.0	6.0	6.1
Tomcat	9.8	9.8	9.8	9.8	10.1	10.2	10.0	10.0	9.9	9.8	9.9	10.0	11.4	11.4	11.4	11.5	11.3	11.5	11.3	11.3
OpenJPA	9.5	9.4	9.5	9.5	9.7	9.7	9.7	9.7	9.8	10.1	9.8	9.8	7.2	7.0	7.2	7.3	6.7	6.8	6.7	6.8
Hibernate ORM	8.9	8.7	8.9	9.3	10.6	11.2	10.6	10.7	6.1	7.0	6.1	4.1	9.5	9.9	9.5	9.5	10.3	9.9	10.3	10.3
Lucene	7.1	6.9	7.1	7.1	9.3	9.3	9.4	9.4	10.8	11.0	10.8	10.8	8.2	8.4	8.2	8.2	10.4	10.4	10.4	10.4
ZooKeeper	9.3	9.2	9.3	9.3	8.8	8.4	8.8	8.8	7.6	7.0	7.6	7.6	7.3	7.3	7.3	7.3	7.9	7.7	7.9	7.9
Avg-Improved	9.0	8.8	9.0	9.1	9.8	9.8	9.8	9.8	8.4	8.6	8.4	8.0	9.1	9.1	9.3	9.7	8.8	8.9	8.8	8.8

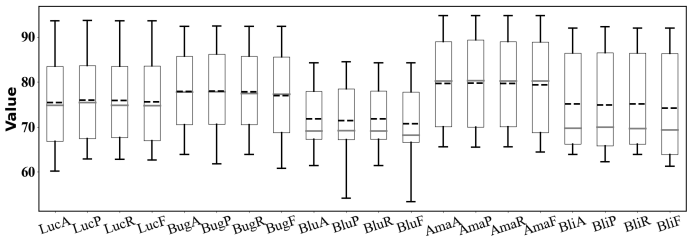
#A represents Accuracy. P represents Precision. R represents Recall. F represents F1-Sore.

RQ1, we also do 10-fold cross-validation over the datasets. The difference is that now our instances only have Pre/Post or Base/Our features. The best-performed RF classifier (in RQ1) is employed during model building.

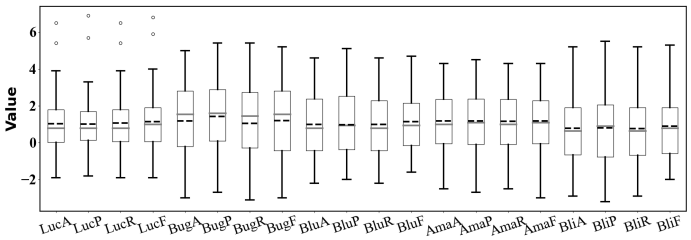
Pre/Post Performance. Fig.2 shows the performance of models built on Pre and Post feature groups respectively over five TRBL techniques. In the figure, Luc, Bug, Blu, Ama, and Bli represent the five TRBL techniques Lucene, BugLocator, Bluir, AmaLgam, and Blizzard. A, P, R, F represents accuracy, precision, recall, F1 score. Hence, for example, BugA means the accuracy of a model in predicting query quality for BugLocator. The dashed black line in the boxplot represents the average line. From Fig.2(a) and Fig.2(b), we can find that all five TRBL techniques except Bluir and Blizzard could obtain an average A, P, R, F of 75% $\sim$ 80% (about 70% for Bluir and Blizzard). And the ranges of the 1st to 3rd quartile values of A, P, R, and F mostly fall into the range of 65% $\sim$ 85%. As for the model performance on Post feature group (Fig.2(c)), we can find that the performance of Post group is similar to or slightly higher than (by up to 1 $\sim$ 2 percent on average) that of Pre group.



(a) Pre



(b) Post



(c) Post - Pre

Fig. 2: Pre vs Post in different TRBL techniques

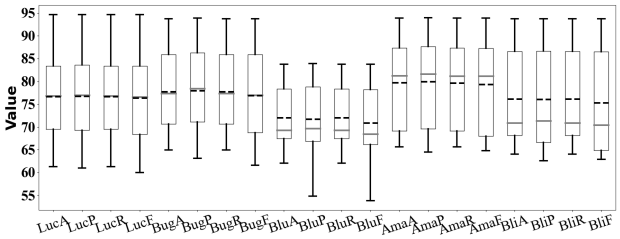
> Finding 4. Our RF models separately built on pre-retrieval and post-retrieval features could also achieve promising prediction results for different TRBL techniques. The average A, P, R, F of models over Pre or Post groups could reach about 70% $\sim$ 80% under different TRBL techniques.

Our/Base Performance. Similar to Pre/Post analysis, we also draw some boxplots to show the performance of our models on Our and Base features in Fig.3. For Our group results in Fig.3(a), we can find that except for Bluir which obtains an average performance of about 70% in terms of A, P, R, F, all other four techniques could obtain an average A, P, R, F of 75% $\sim$ 80%. And the 1st-3rd quartile range of A, P, R, and F mostly falls into the range of 70% $\sim$ 85% (with the 1st quartile valus of Bluir and Blizzard slightly lower than 70%, and the 3rd quartile values of BugLocator and AmaLgam slightly higher than 85%). As for Base group results shown in Fig.3(b)), we can find that the performance of the Base group is, on the whole, worse than that of Our group by about 2 $\sim$ 3 percent on average in terms of A, P, R, F.

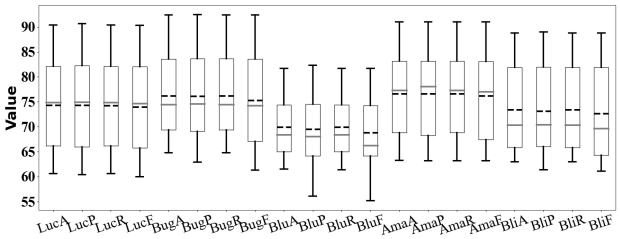
> Finding 5. Our RF models separately built on Our and Base feature groups could obtain promising prediction performance for different TRBL techniques, with the average A, P, R, F being around 75 $\sim$ 80% for Our group and 70 $\sim$ 75% for Base group.

RQ3. Which features are most important in indicating the query quality of a bug report?

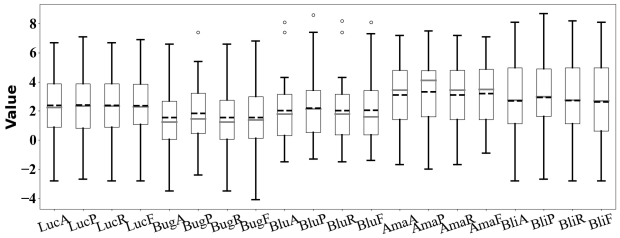
In this RQ, we analyze the importance of individual features for prediction performance. To identify impor-



(a) Our



(b) Base



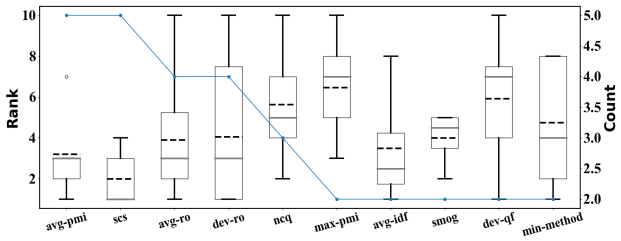
(c) Our - Base

Fig. 3: Our vs Base in different TRBL techniques

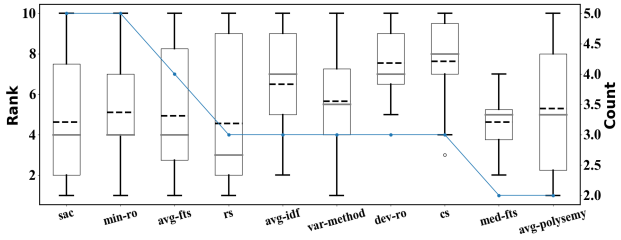
tant features, we first remove redundant features using correlation analysis and redundancy analysis by following existing studies [26]. After reducing the feature set, we build random forest models with 10 cross-validation and assess feature importance. After that, we collect the top 10 most important features based on the importance scores in each project for different TRBL techniques. As mentioned previously in Section II-D, the description and calculation of these features could be found in our shared document at https://goo.su/kHcnCT.

Fig.4 show the distribution of the top 10 most important features over different TRBL techniques. In the figures, the x axis represents the top 10 most important features, the y axis on the left (Rank) means the exact rank of a feature while the y axis on the right (Count) means how many times a feature appeared in the top 10 important feature list for a given TRBL technique over all projects. The boxplot shows the rank distribution of a feature while the blue line indicates the count values.

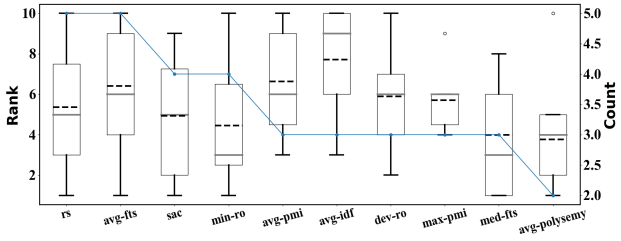
From the figure, we can find that there are 22 features in total that ever appeared in the top 10 most important feature lists on certain projects. Among these features, there are six features that appeared in 4 out of 5 TRBL techniques, they are dev-ro, avg-idf, sac, rs, avg-fts and



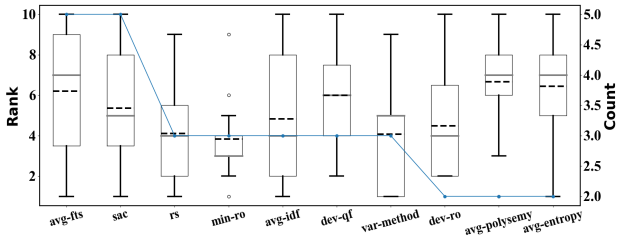
(a) TRBL = Lucene



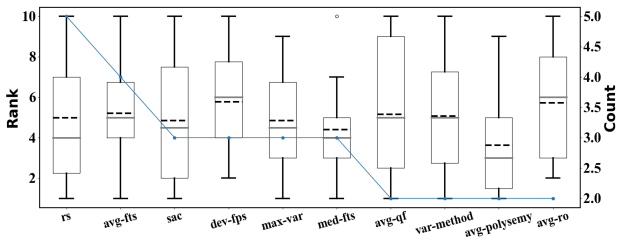
(b) TRBL = BugLocator



(c) TRBL = Bluir



(d) TRBL= AmaLgam



(e) TRBL = Blizzard

Fig. 4: Top-10 most important features under different TRBL techniques

avg-polysemy-value. The Count range (and average Rank) for these features are $2 \sim 4$ (5.3), $2 \sim 3$ (5.3), $2 \sim 5$ (5.1), $2 \sim 5$ (4.9), $2 \sim 5$ (5.7) and $2 \sim 2$ (4.8) respectively. Three features namely min-ro, var-method-num, and med-fts, appeared in three TRBL techniques, with the Count

range (and average Rank) being 2 $\sim$ 5 (4), 2 $\sim$ 3 (4), 2 $\sim$ 3 (3), respectively. In other words, on the whole, there are 9 features that commonly appeared (appearing $>$ 3 out of 5 TRBL techniques) in different TRBL techniques and 7 out of the 9 features have an average rank that lies in the first half of the top 10 lists (<=5).

We analyze which feature group contributes more top-10 most important features over different TRBL techniques. We also find that Post feature group contribute more (or same) top-10 most important feature than that of Pre group, the Count range (and average Count) of top 10 most important features from Pre and Post groups are 3 $\sim$ 5 (3.6) and 5 $\sim$ 7 (6.4). While for the Our/Base groups, we can observe that Our group contributes more top-10 most important features than Base group, the Count range (and average Count) for Our and Base groups are 5 $\sim$ 7 (5.8) and 3 $\sim$ 5 (4.2).

Finding 6.There are 9 features that commonly appeared in the top-10 most important features on experimental projects over different TRBL techniques, they are dev-ro, avg-idf, sac, rs, avg-fts, avg-polysemy-value, min-ro, var-method-num, and med-fts. Moreover, Post group and Our group contribute more top-10 most important features for the five TRBL techniques than that of Pre and Base groups separately.

V. Related Work

1. Query Quality Prediction

Query quality prediction aims to predict the retrieval quality of a query based on the quality of documents returned by a retrieval method against the query. In the field of information retrieval, many query quality estimators have been proposed. These methods can be divided into pre-retrieval methods and post-retrieval methods. Pre-retrieval methods usually predict the retrieval quality of a query before executing it[8][12][11]. Post-retrieval methods usually need to execute the query to obtain an initial result list such as SAC, NCQ [20][21][22].

Within bug localization, the quality of bug reports has attracted many researchers' attention [13] [18]. For example, [42] indicates that bug reports written by even experienced developers can have problems. Some studies measure the quality of bug reports by extracting relevant information from bug reports (such as stack information, code examples, etc.) [1][5]. The mostly related study to ours is Q2P [27], which used 21 pre-retrieval and 7 post-retrieval query features to predict query quality. Based on their features, we integrate more domain-specific features of TRBL tasks and test the generalizability of our approach over different TRBL techniques.

2. Text Retrieval-based Bug Localization

Text retrieval-based bug localization (TRBL) plays a central role in existing approaches to fault localization. Existing TRBL studies have the following four main research directions: replacing a retrieval model with another one to find the model most suitable for bug location, extracting more relevant features to improve model performance, reconstructing low-quality bug reports to further improve the model performance, and applying deep learning methods to facilitate bug location.

For exploring which text retrieval model is suitable for bug localization, there has been a list of studies tested the performance of various text retrieval models such as vector space model (VSM) [1, 2], latent semantic index (LSI) [36, 37], latent Dirichlet allocation (LDA) [34, 35]. As for extracting more relevant features besides textual similarity to improve model performance, BugLocator[2] suggested to use historical bug reports for bug localization. Bluir[5] proposed to consider code structure. Brtracer[1] proposes to make use of exception information. AmaLgam[7] indicates that files that recently caused bugs are likely to cause other errors in the near future. For methods that use query refactoring to improve model performance, Chaparro et al.[32] find that using observed behavior directly instead of a full bug report is effective in finding bugs. Sisman et al.[33] propose a query expansion strategy, which obtains an initial ranking list of code files first by running the location method, and then extracts related terms with higher ranks from the list to expand the bug report. With the continuous development of deep learning technologies, Lam et al.[30] propose an approach that combines a deep neural network with rVSM to extract semantic information from bug reports and source code files. Xiao et al.[31] propose a deep learning method at the character level, where code files and bug reports are represented by characters first, then are passed to CNN for convolution operation, and finally to RNN code for bug location.

VI. Threats to Validity

Internal Threats. In data preparation phase, we use some manually summarized heuristic rules to link a bug report to its corresponding buggy files. We have to admit that the rules may be not 100% correct. To avoid the potential bias, we have tried to manually check the linked data carefully and filtered out wrongly linked bug reports. Another threat is that we re-implemented the Q2P approach based on its description from the original paper [27]. We cannot guarantee that we have 100% correctly implemented Q2P for performance comparison. To avoid the potential threats, we conduct several rounds of code review about Q2P.

External Threats. In this study, all experiments and corresponding analysis are conducted on six open source software (OSS) projects programmed in Java. We cannot guarantee that the arrived conclusions or findings could be applicable to other OSS or industry projects written in Java or other languages. However, considering that these projects are well-known and widely-used projects in practice, plus that they come from different domains and are of different sizes, we believe our experiments on

these projects still shed some light on the capability of our approach in the real world.

VII. Conclusion and Future Work

In this paper, We propose an automated method to predict the query quality of bug reports in information retrieval-based bug localization. We classify bug reports into high-quality and low-quality categories based on whether they can be correctly located by TRBL techniques. Our approach utilizes pre/post-retrieval features from both the general text retrieval domain and the TRBL domain-specific area. By employing a random forest machine learning model, we demonstrate the effectiveness of our approach across different TRBL techniques. Additionally, we investigate the utility of different feature groups in predicting query quality. Last, we identify several common important features that are indicative in predicting the query quality of bug reports. In the future, we plan to develop some reformulation tools for low-quality bug reports once identified by our model.

REFERENCES

[1] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang and H. Mei, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," 2014 IEEE Int. Conf. on Software Maintenance and Evolution., 2014, pp. 181-190.

[2] J. Zhou, H. Zhang and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in Proc. 34th Int. Conf. on Software Engineering., 2012, pp. 14-24.

[3] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in Proc. 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering., 2015, pp. 689–699.

[4] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in Proc. 26th Joint Meeting of European Software Engineering Conf. and Symp. on the Foundations of Software Engineering., 2018, pp 621–632.

[5] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in 28th IEEE/ACM Int. Conf. on Automated Software Engineering., 2013, pp. 345–355.

[6] Lee, Jaekwon, et al, "Bench4bl: reproducibility study on the performance of ir-based bug localization," in Proc. 27th ACM SIGSOFT Int. Symp. on Software testing and analysis., 2018, pp. 61-72.

[7] S. W. Wang and David Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," In Proc. 22nd Int. Conf.on Program Comprehension., 2014, pp.53–63.

[8] B. He and I. Ounis, "Inferring query performance using pre-retrieval predictors," In Proc. of the Symp. on String Processing and Information Retrieval., 2004, pp. 43–54.

[9] Mothe, Josiane, and L. Tanguy, "Linguistic features to predict query difficulty," ACM Conf. on research and Development in Information Retrieval, SIGIR, Predicting query difficulty-methods and applications workshop., 2005. pp. 7-10.

[10] G. A. Miller, "WordNet: An electronic lexical database," MIT press., 1998.

[11] Plachouras, Vassilis, B. He, and I. Ounis. "University of Glasgow at TREC 2004: Experiments in Web, Robust, and Terabyte Tracks with Terrier." TREC., 2004.

[12] Y. Zhao, F. Scholer, and Y. Tsegay, "Effective pre-retrieval query performance prediction using similarity and variability evidence," In Proc. of the European Conf. on Information Retrieval., 2008, pp. 52–64.

[13] Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T, "What makes a good bug report?," in Proc. 16nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering., 2008, pp. 308-318.

[14] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in Proc. IEEE Symp. on visual languages and human-centric computing., 2008, pp. 82–85.

[15] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: A case study on four open source software communities," in Proc. 35th Int. Conf. on Software Engineering., 2013, pp. 1032–1041.

[16] A. Schroter, A. Schr€oter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in Proc. 7th IEEE Work. Conf. on mining software repositories., 2010, pp. 118–121.

[17] W. Weimer, "Patches as better bug reports," in Proc. 5th Int. Conf. on Generative programming and component engineering., 2006, pp. 181–190.

[18] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering., 2007, pp. 34–43.

[19] G. H. Mc Laughlin, "SMOG grading—a new readability formula," J. of Reading., vol. 12, no. 8, pp. 639–646, 1969.

[20] E. Yom-Tov, S. Fine, D. Carmel, and A. Darlow, "Metasearch and federation using query difficulty prediction," in Proc. 28th Int. ACM SIGIR Conf. on Predicting Query Difficulty., 2005.

[21] F. Diaz, "Performance prediction using spatial autocorrelation," in Proc. 30th Int. ACM SIGIR Conf. on Research and development in information retrieval., 2007, pp. 583–590.

[22] Y. Zhou and W. B. Croft, "Ranking robustness: a novel framework to predict query performance," in Proc.15th ACM Int. Conf. on Information and knowledge management., 2006, pp. 567–574.

[23] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report," in Proc. 14th Int. Conf. Mining Softw.Repositories., 2017, pp. 170–181.

[24] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" IEEE Trans. on Software Engineering., vol. 34, no. 2, pp. 181–196, 2008.

[25] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in Proc. 7th IEEE Work. Conf. on mining software repositories., 2010, pp. 1–10.

[26] L. Bao and X. Xia, et al, "A large scale study of long-time contributor prediction for github projects," IEEE Trans. on Software Engineering., vol. 47, no. 6, pp. 1277-1298, 2019.

[27] C. Mills, G. Bavota and S. Haiduc, et al, "Predicting query quality for applications of text retrieval to software engineering tasks," ACM Trans. on Software

Engineering and Methodology., vol. 26, no. 1, pp. 1-45, 2017. Trans. on Software Engineering., vol. 46, no. 8, pp. 836-862, 2018.

[28] D. Carmel and E. Yom-Tov, "Estimating the query difficulty for information retrieval," Syn Lect, on Information Concepts., vol. 2, no. 1,,1-89, 2010.

[29] M. Zhou and A. Mockus, "Who will stay in the FLOSS community? Modeling participant's initial behavior," IEEE Trans. on Software Engineering., vol. 41, no. 1, pp. 82–99, 2015.

[30] A. N. Lam, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in Proc. 5th Int. Conf. on Program Comprehension., 2017, pp. 218–229.

[31]] Y. Xiao, J. Keung, "Improving bug localization with character-level convolutional neural network and recurrent neural network," In 25th. Asia-Pacific Software Engineering Conference., 2018, pp. 703-704.

[32] O. Chaparro, "Improving bug reporting, duplicate detection, and localization," In Proc. 39th Int. Conf. on Software Engineering., 2017, pp. 421-424.

[33] B. Sisman, A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," In Proc. 20th Working Conf. on Mining Software Repositories., 2013, pp. 309-318.

[34] A. T. Nguyen, T. T. Nguyen, J. SI-Kofahi, H. V. Nguyen, T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," In Proc. 26th Int. Conf. on Automated Software Engineering., 2011, pp. 263-272.

[35] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," In Proc. 15th Working Conf. on Reverse Engineering., 2008, pp. 155-164.

[36] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, "Indexing by latent semantic analysis," Journal of the American society for information science., vol. 41, no, 6, 1990, pp. 391-407.

[37] T. Hofmann, "Probabilistic latent semantic indexing," In Proc. 22th Int. Conf. on Research and Development in Information Retrieval., 1999, pp. 50–57.

[38] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Trans. on Software Engineering., vol. 20, no. 6, pp. 476-493, 1994.

[39] L. Naish, Neelofar, K. Ramamohanarao, "Multiple bug spectral fault localization using genetic programming," In Proc. 24th. Australasian Software Engineering Conference., 2015, pp. 11-17.

[40] H. J. Lee, L. Naish, K. Ramamohanarao, "Effective software bug localization using spectral frequency weighting function," In Proc. 34th. Annual Computer Software and Applications Conference., 2010, pp. 218-227.

[41] S. Cronen-Townsend, Y. Zhou, and W. B. Croft, "A language modeling framework for selective query expansion," Massachusetts Univ Amherst Center. for Intelligent Information Retrieval., 2004.

[42] W. Q. Zou,J. X. Zhang,X. W. Zhang, L. Chen, J. F. Xuan, "Survey of Research on Bug Report Quality," Journal of Software., 2022.