International Journal of Software Engineering
and Knowledge Engineering
Vol. 27, No. 6 (2017) 869–896
© World Scientific Publishing Company
DOI: 10.1142/S0218194017500322

PRST: A PageRank-Based Summarization Technique for Summarizing Bug Reports with Duplicates

He Jiang[*,§], Najam Nazar[†,¶], Jingxuan Zhang[*,‖],
Tao Zhang[‡,**] and Zhilei Ren[*,††]

[*]*Key Laboratory for Ubiquitous Network and*
Service Software of Liaoning Province
School of Software
Dalian University of Technology
Dalian, P. R. China

[†]*Faculty of Information Technology, Monash University, Australia*

[‡]*College of Computer Science and Technology*
Harbin Engineering University, Harbin, China

[§]*jianghe@dlut.edu.cn*
[¶]*najam.nazar@monash.edu*
[‖]*jingxuanzhang@mail.dlut.edu.cn*
[**]*cstzhang@hrbeu.edu.cn*
[††]*zren@dlut.edu.cn*

Received 25 July 2015
Revised 9 January 2016
Accepted 23 August 2016

During software maintenance, bug reports are widely employed to improve the software project's quality. A developer often refers to stowed bug reports in a repository for bug resolution. However, this reference process often requires a developer to pursue a substantial amount of textual information in bug reports which is lengthy and tedious. Automatic summarization of bug reports is one way to overcome this problem. Both supervised and unsupervised methods are effectively proposed for the automatic summary generation of bug reports. However, existing methods disregard the significance of duplicate bug reports in summarizing bug reports. In this study, we propose a PageRank-based Summarization Technique (PRST), which utilizes the textual information contained in bug reports and additional information in associated duplicate bug reports. PRST uses three variants of PageRank-based on Vector Space Model (VSM), Jaccard, and WordNet similarity metrics. These variants are utilized to calculate the textual similarity of the sentences between the master bug reports and their duplicates. PRST further trains a regression model and predicts the probability of sentences belonging to the summary. Finally, we combine the values of PageRank and regression model scores to rank the sentences and produce the summary for the master bug reports. In addition, we construct two corpora of bug reports and duplicates, i.e. MBRC and OSCAR. Empirical results suggest that PRST outperforms the state-of-the-art method BRC in terms of Precision, Recall, F-score, and

[§]Corresponding author.

Pyramid Precision. Meanwhile, PRST with WordNet achieves the best results against PRST with VSM and Jaccard.

Keywords: Duplicate bug reports; summarization; supervised learning; PageRank.

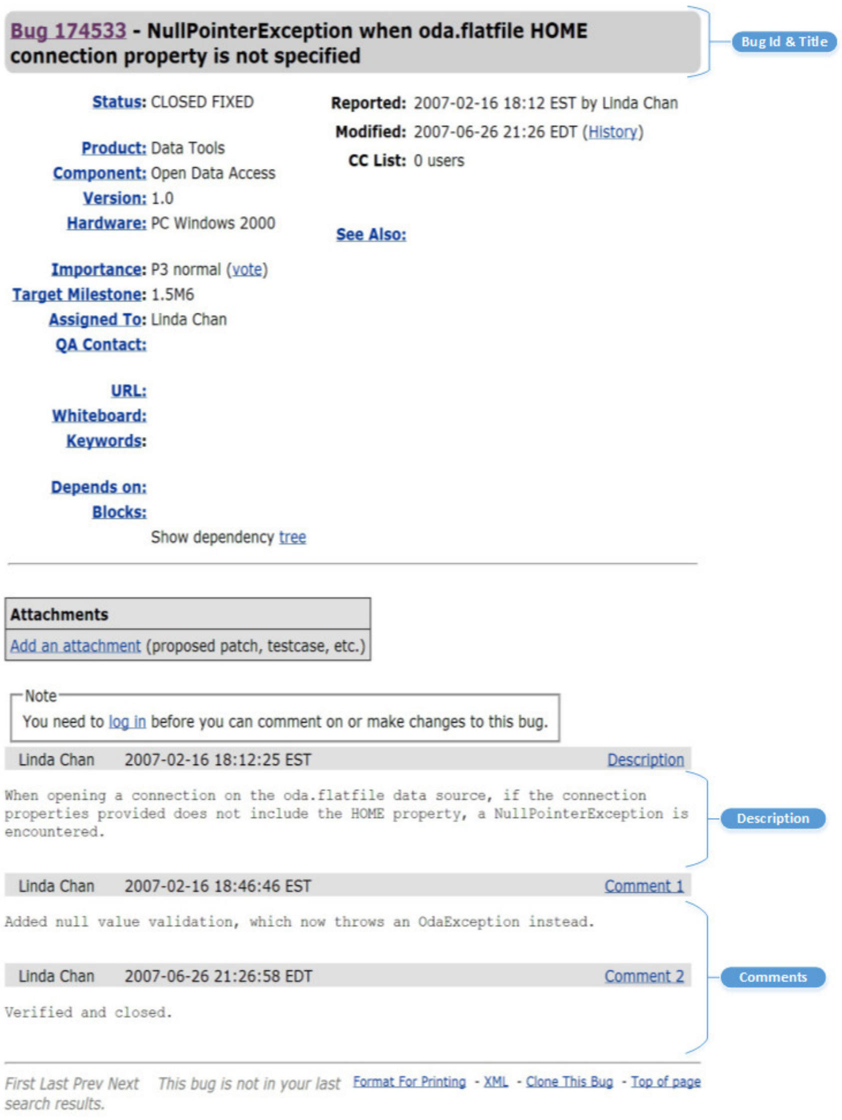
1. Introduction

Bug reports are valuable assets in software development projects. A bug report often resembles a recorded, sequentially ordered conversation from several people where each message is composed of multiple sentences. Usually, a bug report contains the title of the problem, bug items that provide general information such as product and component of the bug report, a description written by the reporter, and comments by other users or developers. The description in a bug report narrates the production of unusual behavior, while comments record the debate between the developers about the bug and its resolution. Figure 1 exhibits a typical bug report from the Eclipse bug repository.[a] As the bug is resolved over the period, this conversation becomes lengthy, tedious, and difficult to understand for developers. It is time consuming for developers to consult and understand these lengthy bug conversations. Thus, a brief and accurate textual representation of lengthy bug conversations eases the arduous effort of understanding bug reports.

While analyzing the state-of-the-art extractive and abstractive techniques for summarizing bug reports, it has been observed that these techniques hardly achieve the precision of 60%, when tested on publicly available corpora. Rastkar *et al.* [1] applied pre-existing supervised learning methods for generating extractive summaries of bug reports. They trained a logistic regression classifier *BRC* and predicted the probability of each sentence belonging to the summary. By selecting a set of top ranked sentences from the original bug report, a succinct summary of bug report can be achieved. When they test this method on an open bug report corpus, they achieve the precision of 57%. In contrast, Mani *et al.* [2] applied unsupervised learning techniques on the same BRC corpus. They introduced a noise reduction mechanism, which classified sentences heuristically into different types for generating concise summaries. Such methods mainly focused on resolving or improving a learning model but neglected the importance of other components or characteristics of the bug repository such as duplicate bug reports.

Moreover, in a recent effort, Rastkar *et al.* [3] found that concise summaries of original bug report can help developers find duplicate bug reports effectively. Using the same intuition, we hypothesize if the bug report summaries can effectively help detecting duplicate bug reports then the information contained in duplicate bug reports can also be utilized for generating better summaries of bug reports. Therefore, in order to overcome the aforementioned challenges (discussed in the last paragraph) and test our hypothesis, we propose a novel technique to accurately generate summaries of master bug reports by utilizing the textual information

[a] bugs.eclipse.org/bugs, verified 25/07/15.


Bug 174533 - NullPointerException when oda.flatfile HOME connection property is not specified

Status: CLOSED FIXED
Reported: 2007-02-16 18:12 EST by Linda Chan
Modified: 2007-06-26 21:26 EDT (History)
CC List: 0 users

Product: Data Tools
Component: Open Data Access
Version: 1.0
Hardware: PC Windows 2000

See Also:

Importance: P3 normal (vote)
Target Milestone: 1.5M6
Assigned To: Linda Chan
QA Contact:

URL:
Whiteboard:
Keywords:

Depends on:
Blocks:
Show dependency tree

Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Linda Chan 2007-02-16 18:12:25 EST Description

```
When opening a connection on the oda.flatfile data source, if the connection
properties provided does not include the HOME property, a NullPointerException is
encountered.
```

Linda Chan 2007-02-16 18:46:46 EST Comment 1

```
Added null value validation, which now throws an OdaException instead.
```

Linda Chan 2007-06-26 21:26:58 EDT Comment 2

```
Verified and closed.
```

First Last Prev Next This bug is not in your last search results. Format For Printing - XML - Clone This Bug - Top of page

Fig. 1. An example of a typical bug report. Bug # 174533 from Eclipse bug repository.

contained in duplicate bug reports. As each bug report resembles a conversation, text summarization techniques have been effectively applied earlier to summarize bug report conversations. Therefore, the textual information of duplicate bug reports can be utilized effectively for summarizing master bug reports as it provides additional information.

In this study, we propose PRST which summarizes master bug reports by utilizing the information contained in the associated duplicate bug reports. To simulate this phenomenon, we modify the existing BRC[b] [1] corpus to map duplicate bug reports and meanwhile construct a new bug report corpus, *OSCAR*,[c] that contains master and duplicate bug reports mapped together. In total, the modified BRC corpus i.e. *MBRC* contains 28 bug reports with 935 sentences, whereas *OSCAR* contains 59 bug reports with 1399 sentences. Both corpora are extracted from Eclipse, Mozilla,[d] KDE,[e] and Gnome,[f] open source projects. To calculate the textual similarity between sentences in bug reports, we apply three variants of PageRank based on similarity metrics i.e. *WordNet* [4], *Vector Space Model* (*VSM*) [5], and *Jaccard* [6] methods. In addition, we employ supervised learning methods to train a logistic regression model and predict the probability of each sentence belonging to a summary. Thus, we can form a network and apply PageRank algorithm. After merging the PageRank and regression scores, the high ranking sentences are selected as summary sentences. By utilizing four widely used evaluation metrics, *Precision, Recall, F-Score*, and *Pyramid Precision*, PRST improves the Precision of the state-of-the-art method *BRC* by 9%. This proves that the information contained in duplicate bug reports can help producing better summaries of bug reports than existing methods. Our contributions in this paper are as follows:

(1) To the best of our knowledge, it is the first work to introduce the valuable information of duplicate bug reports to automatically summarize the master bug reports. Our proposed technique, PRST, applied three variants of PageRank by introducing the textual similarity of bug reports and their duplicates that could generate better bug report summaries. The experimental results show that the novel technique can improve the summarization accuracy.
(2) We constructed a new corpus (i.e. OSCAR), which consists of 59 bug reports containing master and duplicate bug reports. We invited 10 graduate students from the same laboratory, who have good programming experience for annotating the corpus. OSCAR corpus is publicly available and future researchers can benefit from it.
(3) Extensive experiments were carried out on OSCAR and MBRC corpora to substantiate the advantages of the proposed PRST over the existing methods.

The paper is organized as follows: We present the motivation behind our proposed idea in Sec. 2. Section 3 describes our summarization technique and the main components. Section 4 details the insight into the experimental design, while Sec. 5 discusses the experimental results and threats to the validity of our work in detail. In Sec. 6, we provide a brief literature review in the bug report and source code repositories. Section 7 concludes our paper.

[b] BRC corpus, in its original form, can be accessed at http://www.cs.ubc.ca/cs-research/software-practices-lab/projects/summarizing-software-artifacts, verified 25/07/15.
[c] OSCAR corpus is publicly available at http://oscar-lab.org/paper/prst/corpus.htm, verified 25/07/15.
[d] bugzilla.mozilla.org/, verified 25/07/15.
[e] bugs.kde.org/, verified 25/07/15.
[f] bugzilla.gnome.org. verified 25/07/15.

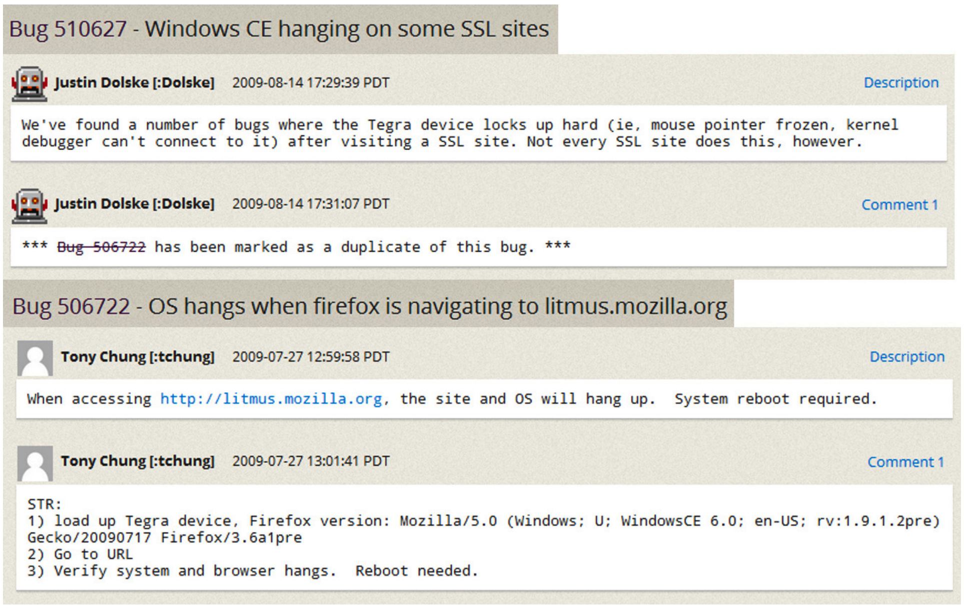
2. Motivation

Academic researchers have conducted studies to detect [7], automate [8], prioritize [9] and triage [10] duplicate bug reports. However, they have not used duplicate bug reports for summarization task yet. The motivation of our work is based on the notion that the master bug reports can be effectively summarized by utilizing the textual information between the master and the duplicate bug reports. The reason is that the duplicate bug reports are similar to master bug reports as both bug reports discuss the same problem in different words. Moreover, duplicate bug reports can provide additional useful information that can be further utilized for summarization process. Here, we explain our motivation using the following example:

Let us consider a bug report # 510627 and its duplicate bug report # 506722, extracted from the Mozilla Bug Repository[g] as an example (Fig. 2).

In the bug report # 510627, Justine Dolske described the problem in the following words

> "... *We've found a number of bugs where the Tegra device locks up hard (ie, mouse pointer frozen, kernel debugger can't connect to it) after visiting a SSL site. Not every SSL site does this, however*...".



Bug 510627 - Windows CE hanging on some SSL sites

Justin Dolske [:Dolske] 2009-08-14 17:29:39 PDT Description

```
We've found a number of bugs where the Tegra device locks up hard (ie, mouse pointer frozen, kernel
debugger can't connect to it) after visiting a SSL site. Not every SSL site does this, however.
```

Justin Dolske [:Dolske] 2009-08-14 17:31:07 PDT Comment 1

```
*** ~~Bug 506722~~ has been marked as a duplicate of this bug. ***
```

Bug 506722 - OS hangs when firefox is navigating to litmus.mozilla.org

Tony Chung [:tchung] 2009-07-27 12:59:58 PDT Description

```
When accessing http://litmus.mozilla.org, the site and OS will hang up.  System reboot required.
```

Tony Chung [:tchung] 2009-07-27 13:01:41 PDT Comment 1

```
STR:
1) load up Tegra device, Firefox version: Mozilla/5.0 (Windows; U; WindowsCE 6.0; en-US; rv:1.9.1.2pre)
Gecko/20090717 Firefox/3.6a1pre
2) Go to URL
3) Verify system and browser hangs.  Reboot needed.
```

Fig. 2. An example of Master and its corresponding duplicate bug report. (The beginning parts of Bug # 510627 and Bug # 506722 from Mozilla bug repository).

[g]bugzilla.mozilla.org, verified 25/07/15.

The above description mentions that the mouse pointer and kernel debugger hang when SSL sites are visited on a Tegra device. However, it does not happen on visiting all SSL sites.

Similarly, its duplicate bug report # 506722, reported by Tony Chung, describes the same issues as follows:

> " . . . *STR: 1) load up Tegra device, Firefox version: Mozilla/5.0 (Windows; U; WindowsCE 6.0; en-US; rv: 1.9.1.2pre) Gecko/20090717 Firefox/3.6a1pre 2) Go to URL 3) Verify system and browser hangs* . . . ".

The excerpt from bug # 506722 notifies that the OS hangs while visiting SSL-based websites.

Even though the textual description differs in both examples, the two bug reports reported the similar problem. Furthermore, by reading the descriptions of these bug reports, we found that these reports often use terms or words that are similar in nature. For example, both bug reports share some common words, such as 'Tegra', 'hangs', and 'site' and these words can be included in a final summary. In total, bug report # 510627 has five duplicate bug reports including # 506722, # 504970, # 508478, # 510419, and # 502230, and these five duplicate bug reports describe the same problem differently.

From the aforementioned example, we can see that the textually similar information in both bug reports as well as the supported information in duplicates can be employed effectively for automatic summarization of master bug reports. We formalized this intuition by investigating more bug reports and proposing a novel technique (discussed in next Section) for generating bug reports summaries.

3. Methodology

Figure 3 outlines the proposed PRST. To produce the natural language summaries of master bug reports, we first collected master bug reports and their duplicates from open source bug report repositories (corpus). Each bug report in a corpus contains a

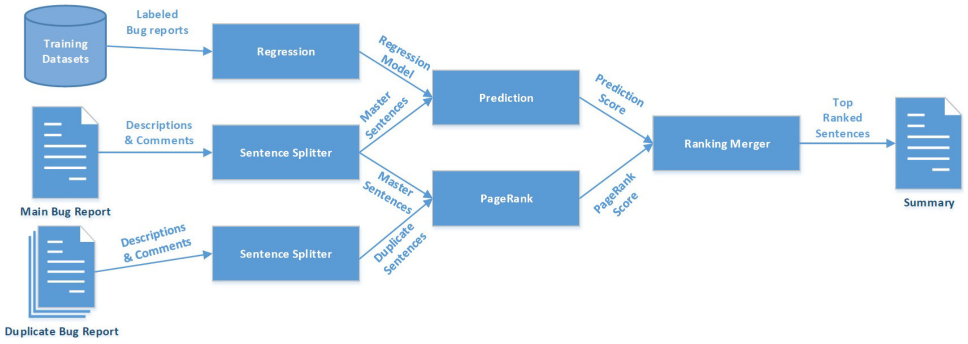


Fig. 3. PRST Technique for bug report summarization.

bug description and its related comments. After creating the bug report corpus, annotation methods were applied to label the corresponding bug reports (training set). The textual contents such as descriptions and comments of the master and duplicate bug reports were broken into sentences through a *Sentence Splitter*. Second, these sentences were passed to the ranking module, where PageRank algorithm ranked these sentences by applying different similarity measures. Next, the supervised machine learning prediction model estimated the probability score of sentences in master bug reports, when passed with the training set. Further, a ranking merger combined the scores attained through PageRank and supervised learning. Sentences were further sorted on the basis of final scores achieved through the ranking merger component. In the end, top ranked sentences were selected as the summary sentences of the master bug report.

In the subsequent sections, we provide details of each component of PRST. These components include *Sentence Splitter*, *PageRank*, *Regression*, and *Predication* and *Ranking Merger*.

3.1. *Sentence splitter*

While constructing a bug report summary, we could decide at the specified granularity level, whether the piece of text in the input document should be present in the output summary or not [2]. The granularity can be at the level of a word, a sentence, or a paragraph. A word level granularity being too small creates reading difficulty while a paragraph-level granularity includes redundant information. Therefore, we selected a sentence-level granularity for summary construction in this study.

We regarded the textual information contained in a title, description, and comments of bug reports to form a summary. *LingPipe Toolkit*[h] (Alias-i LingPipe Tokenizer) was used to split descriptions and comments into sentences. Lingpipe is a toolkit that utilizes computational linguistics for text processing. Although it could perform other tasks, such as word analysis, we mainly employed it for splitting textual data into sentences. It took descriptions and comments of bug reports as an input and produced sentences as an output.

Pre-processing Steps: Furthermore, as a standard, we applied pre-processing techniques i.e. *tokenization*, *stop words removal*, and *stemming* at this level. Tokenization process breaks the text into words, phrases, symbols or other meaningful elements. The stop word removal process filters out unnecessary words from the tokens while, stemming is used to transform words into their root forms. For example, 'watch' is a root form for the words 'watching' or 'watched'.

3.2. *PageRank module*

In this module, we applied three variants of PageRank algorithm based on *VSM*, *Jaccard*, and *WordNet* metrics to measure the similarity among sentences. This

[h]http://alias-i.com/lingpipe/, verified 25/07/15.

module generated the probability scores for each variant of PageRank when sentences attained through *Sentence Splitter* were passed as an input. In the end, sentences were sorted in descending order.

The PageRank algorithm is used to calculate the importance of web pages connected or linked together on the internet. It assigns a numerical weight to each node on the web page with the purpose of measuring the relative 'importance' of the set of connected links [11, 12]. The rank value indicates an importance of a particular page. If a page has many inbound links, it has a higher rank value. Similarly, if a page has no inbound link, it is not important at all. Intuitively, *Google* is an important page, reflected by the fact that many pages point to it. Likewise, pages that are pointed from *Google* are themselves important. Here, nodes are constructed with sentences and edges are established using similarity score among terms i.e. words or sentences [12].

In our work, we modified the original PageRank algorithm slightly to fit our scenario, since our main purpose was to measure the importance of sentences in master and duplicate bug reports. While performing PageRank on bug reports, we connected sentences from the master and duplicate bug reports with each other as well as the sentences that are related to master and duplicate bug reports only. Sentences are linked with each other in the same bug report as well as among master and duplicate bug reports. When there were two sentences with similarity scores greater than 0 between the master and the duplicate as well as between each other, two directed edges were formed between the two sentences. A PageRank web graph was established when all sentences were processed. In the end, sentences were sorted based on the ranking score achieved through PageRank. The high ranking sentences were important sentences and included in the summary.

As mentioned in Sec. 1, we employed three variants of PageRank algorithm for measuring textual similarity among bug report sentences. These similarity measures were *VSM*, *Jaccard*, and *WordNet*. In subsequent sections, we discuss these measures briefly.

3.2.1. *VSM*

The *Vector Space Model* (*VSM*) is a simple algebraic model based on the term-document matrix of a corpus [13]. VSM represents documents by their column vector in the term-document matrix: a vector containing the weights of the words present in the document, or 0s otherwise [13]. A common way to compute the similarity of two vectors in VSM is using the *Cosine Similarity* measure. *Cosine Similarity* is defined as the inner product of the two vectors (sum of the pairwise multiplied elements) divided by the product of their vector lengths [5]. It is the most commonly used metrics for calculating the similarity between textual data. Based on this information, we selected the *Cosine Similarity* metrics for calculating the similarity between sentences of bug reports.

After applying pre-processing steps, we computed the *term frequency − inverse document frequency* (*tf-idf*) values on term vectors. The *tf-idf* is a popular vector weighting method for discovering term relevancy in a corpus. It assigns a high weight to a term, if it occurs frequently in the document but rarely in the whole collection [5]. *Tf-idf* is scaled logarithmically as:

$$\omega_{ik} = 1 + \log(f_{ik}) \times \log\frac{N}{n_i},$$

where

- f_{ik} is the term frequency of the ith term in sentence k.
- N stands for the total number of sentences.
- n_i denotes the number of sentences where the ith term occurs at least once.

Using this vector space model where each vector represents a sentence, we calculated the cosine similarity between the vector representations of two sentences as follows.

$$\cos(\overrightarrow{\omega_q}, \overrightarrow{\omega_k}) = \frac{\overrightarrow{\omega_q} \cdot \overrightarrow{\omega_k}}{|\overrightarrow{\omega_q}||\overrightarrow{\omega_k}|} = \frac{\sum_{i=1}^{n}\omega_{iq} \times \omega_{ik}}{\sqrt{\sum_{i=1}^{n}\omega_{iq}^2}\sqrt{\sum_{i=1}^{n}\omega_{ik}^2}},$$

where

- $\overrightarrow{\omega_q}$ and $\overrightarrow{\omega_k}$ are the vectors that represent two sentences q and k, respectively.
- ω_{iq} and ω_{ik} denote the weight of the ith term in sentences q and k, respectively.
- n is the size of the word set.

3.2.2. *Jaccard*

The Jaccard measures the similarity between finite sample sets using the collection of words called a *bag of words*. It is defined as the size of the intersection of the sample sets divided by the size of the union of the sample sets. If we denote the sets of terms S from two sentences as A_S and B_S, the Jaccard coefficient is calculated as:

$$J(A_S, B_S) = \frac{|A_S \bigcap B_S|}{|A_S \bigcup B_S|},$$

where

- A_S and B_S are sets of terms S in sentences A and B.

3.2.3. *WordNet*

WordNet is a semantic web covering a wide range of English language words, where nouns, verbs, adjectives, and adverbs are grouped into sets of synonyms known as *synsets* [14]. These synonyms or near synonyms are connected together to represent the same meanings. One way to measure the sentence relevance between the sentences of bug reports in both corpora is to apply WordNet. Using synsets, the similarity between two sentences can be measured effectively. Given two sentences, each sentence is tokenized and tagged. Next, after stemming, most appropriate senses of words in sentences are gathered through synsets — called *disambiguation*. Lastly, the similarity of the sentences based on the similarity of the pairs of words is computed. The higher the score, the more similar the meaning of the two sentences.

WordNet is a freely available software package and there are many online tools available for WordNet computation. In our study, we employed *WordNetDotNet*[i] framework available at GoogleCode to measure the similarity and generate synsets.

3.3. *Regression module*

By using the training set which contains the annotated data, we trained our classifier. Our classifier is a logistic regression classifier and instead of generating an output of zero or one, it generates the probability of each sentence belonging to the summary. We extracted 24 sentence features inspired by the technique proposed by Rastkar *et al.* [1]. The values of these features for each sentence were used to compute the probability of the sentence being a part of the summary.

We used *LibLinear Toolkit*[j] to implement our logistic regression classifier. It takes the features and the class label of each sentence as an input, and outputs a logistic regression model.

3.4. *Prediction module*

As supervised learning method trains a statistical model and executes the model on a remaining set of the information, this component serves as a prediction for the model that we trained in Sec. 3.3.

The probability of each sentence of the master bug reports was generated by infusing the features of each sentence into the trained model. In the end, sentences were sorted based on the probabilities and higher-probability sentences were considered as important sentences.

3.5. *Ranking merger*

The ranking merger aims to merge PageRank and regression scores. We calculated it using the following equation.

$$f_j = \alpha \times PR_j + (1 - \alpha) \times LR_j,$$

where PR_j is a PageRank value of sentence j while LR_j is the Regression value for sentence j. We normalized both PageRank (PR_j) and Regression values (LR_j) in the range of 0 to 1, before adding them.

[i] WordNetDotNet can be accessed at http://code.google.com/p/wordnetdotnet/, verified 25/07/15.
[j] http://www.csie.ntu.edu.tw/~cjlin/liblinear/, verified 25/07/15.

Alpha Weighting Factor: α is a weighting factor ($0 \leq \alpha \leq 1$). The parameter α adjusts weights between the Pagerank and regression values. The value of α can be set and adjusted empirically to achieve the best results for all three variants of PRST. We employed the *Trial and Error* procedure for adjusting the alpha weighting factor. In our experiment, for every iteration, we incremented α by 0.01 until it reached 1. α as a weighting factor has been effectively utilized by research community in different research domains (e.g. Bug Localization [15]) and our idea of utilizing α weighting factor is also inspired by the Zhou *et al.*'s [15] work.

Top Ranked Sentences: Finally, to form the summary, we sorted the sentences into a list based on the *Ranking Merger* probability scores in descending order. Starting from the beginning of this list, we chose to target the top 25% sentences as summary sentences since this value was close to the percentage of golden summary sentences. This idea was inspired by the earlier work done by Rastkar *et al.* [1]. They chose to target 25% of the bug report word count as it was close to the word count percentage of the gold standard summary. In our study, a cut-off point for summary selection was 25.7%. Therefore, we selected top 25% sentences as summary sentences i.e. *top ranked sentences*.

Moreover, we generated summaries for master bug reports only; similar sentences from duplicate bug reports were not included in the final selection of summary sentences. Algorithm 1 provides the pseudo-code for our PRST Technique.

4. Experimental Design

In the following subsections, we discuss the data acquisition, annotation, conversational features, research questions, the BRC algorithm, evaluation methods and measures, which we have utilized in our study.

Algorithm 1 Pseudo-code for PRST Technique

Input: Master bug report with duplicates, Training corpora

Output: Top ranked sentences which can form summary for master bug report

```
1: Model = TrainRegressionModel(Labeled bug reports in training corpora);
2: Msentences = SentenceSpliter(master bug report);
3: Dsentences = SentenceSpliter(duplicate bug reports);
4: PageRankScore = PageRank(Msentence, Dsentence);
5: PredictScore = Predict(Model, Msentence);
6: for all sentence in Msentence do
7:     FinalScore = $\alpha\times$ PageRankScore + (1-$\alpha$)$\times$ PredictScore;
8: end for
9: TopRankedSentence = RankSentence(FinalScore);
10: return TopRankedSentences;
```

4.1. *Data acquisition*

There existed a bug report corpus, *BRC* [1] consisting of 36 bug reports taken from four different projects, namely Mozilla, Eclipse, Gnome, and KDE. These 36 bug reports were selected randomly, 9 from each open source project. This corpus was well labeled and used by researchers in academia previously. However, it was unsuitable for our scenario due to the fact that we needed master bug reports and their duplicate bug reports mapped together in a single corpus. There was no connection between master and duplicate bug reports in the existing BRC corpus. Therefore, we modified the existing BRC corpus, which we shall refer to as *MBRC* onward, and built a new corpus of bug reports called *OSCAR*, to map master and duplicate bug reports.

In the following sections, we describe the reconstruction of BRC corpus and the construction of OSCAR corpus.

4.1.1. *Reconstruction of BRC corpus (MBRC corpus)*

To reconstruct the BRC corpus, we examined each bug report if it contained duplicate bug reports or not. First, we read the information in all bug reports and searched text containing ****has been marked as a duplicate of this bug**** to obtain the list of duplicate bug reports. If this phrase existed in the text and the duplicate bug report existed in a corpus as well, we retained both of them. Second, if the master bug report contained the text about duplication but the required duplicate bug was not part of the actual BRC, we added the duplicate bug in MBRC corpus (the modified BRC corpus). For instance, the bug report #66526 (extracted from KDE bug repository) in original BRC has a duplicate bug 71479 but this duplicate bug does not exist in original BRC. Therefore, we retrieved the required information from its corresponding bug report repository. Furthermore, if a bug report has more than one duplicate bug report, we extracted the required information about all related duplicate bug reports to that master bug report from the web as well. This process continued until all required information was retrieved and stored. In total, there were 28 bug reports in the modified version of BRC corpus (MBRC), containing 9 master bug reports and 19 duplicate bug reports.

4.1.2. *Construction of OSCAR corpus*

We constructed a new corpus called OSCAR to obtain more generalized results. Our corpus OSCAR is similar to MBRC corpus in the sense that both corpora exhibit same structure. In addition, OSCAR contains the master bug reports and associated duplicate bug reports together. As in BRC, MBRC and OSCAR as well, we selected bug reports from four open source projects, Eclipse, Mozilla, KDE, and Gnome. We considered the following requirements while selecting bug reports for OSCAR corpus:

(1) It contains duplicate bug reports and there should be a mapping (relationship) between master and their respective duplicates,

(2) It does not contain a large number of code segments and stack traces as this content may be used but not typically read by developers, and
(3) Bug report length should vary, covering all circumstances.

In OSCAR, there are 59 bug reports including 19 master bug reports and 40 duplicate bug reports.

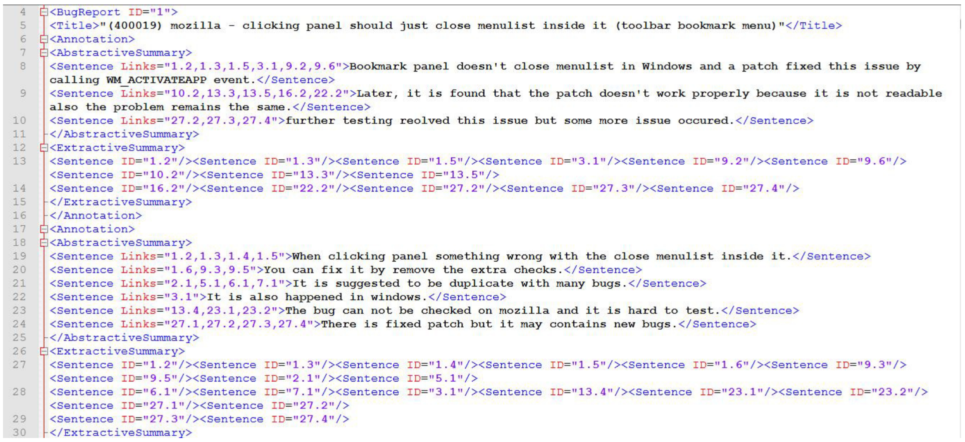
Table 1 lists averages, minima, and maxima of number of comments, duplicates, sentences in master and duplicate bug reports for both OSCAR and MBRC corpora.

Table 1. Details of the subjects for OSCAR and MBRC Corpora.

	OSCAR			MBRC		
	Avg	Min	Max	Avg	Min	Max
# of comments	13.40	4	28	16.37	9	23
# of duplicates	2.05	1	7	2.50	1	6
# of sentences in master	44.25	12	95	71.62	50	104
# of sentences in duplicate	13.95	2	52	15.27	6	41

4.2. *Data annotation*

To annotate the OSCAR corpus, we recruited 10 postgraduate students from the School of Software at the Dalian University of Technology. On average, the annotators have more than four years of experience in programming. Currently, they are pursuing academic research in Mining Software Repositories (MSR). Therefore, it was easy for them to read and understand the bug reports thoroughly to perform annotation. All bug reports were divided among participants randomly and each bug report was annotated by three different participants as a whole to get more consistent results. Figure 4 shows an example of a part of an annotated bug report.



```
<BugReport ID="1">
<Title>"(400019) mozilla - clicking panel should just close menulist inside it (toolbar bookmark menu)"</Title>
<Annotation>
<AbstractiveSummary>
<Sentence Links="1.2,1.3,1.5,3.1,9.2,9.6">Bookmark panel doesn't close menulist in Windows and a patch fixed this issue by
calling WM_ACTIVATEAPP event.</Sentence>
<Sentence Links="10.2,13.3,13.5,16.2,22.2">Later, it is found that the patch doesn't work properly because it is not readable
also the problem remains the same.</Sentence>
<Sentence Links="27.2,27.3,27.4">further testing reolved this issue but some more issue occured.</Sentence>
</AbstractiveSummary>
<ExtractiveSummary>
<Sentence ID="1.2"/><Sentence ID="1.3"/><Sentence ID="1.5"/><Sentence ID="3.1"/><Sentence ID="9.2"/><Sentence ID="9.6"/>
<Sentence ID="10.2"/><Sentence ID="13.3"/><Sentence ID="13.5"/>
<Sentence ID="16.2"/><Sentence ID="22.2"/><Sentence ID="27.2"/><Sentence ID="27.3"/><Sentence ID="27.4"/>
</ExtractiveSummary>
</Annotation>
<Annotation>
<AbstractiveSummary>
<Sentence Links="1.2,1.3,1.4,1.5">When clicking panel something wrong with the close menulist inside it.</Sentence>
<Sentence Links="1.6,9.3,9.5">You can fix it by remove the extra checks.</Sentence>
<Sentence Links="2.1,5.1,6.1,7.1">It is suggested to be duplicate with many bugs.</Sentence>
<Sentence Links="3.1">It is also happened in windows.</Sentence>
<Sentence Links="13.4,23.1,23.2">The bug can not be checked on mozilla and it is hard to test.</Sentence>
<Sentence Links="27.1,27.2,27.3,27.4">There is fixed patch but it may contains new bugs.</Sentence>
</AbstractiveSummary>
<ExtractiveSummary>
<Sentence ID="1.2"/><Sentence ID="1.3"/><Sentence ID="1.4"/><Sentence ID="1.5"/><Sentence ID="1.6"/><Sentence ID="9.3"/>
<Sentence ID="9.5"/><Sentence ID="2.1"/><Sentence ID="5.1"/>
<Sentence ID="6.1"/><Sentence ID="7.1"/><Sentence ID="3.1"/><Sentence ID="13.4"/><Sentence ID="23.1"/><Sentence ID="23.2"/>
<Sentence ID="27.1"/><Sentence ID="27.2"/>
<Sentence ID="27.3"/><Sentence ID="27.4"/>
</ExtractiveSummary>
```

Fig. 4. A screenshot of the annotation for the bug # 400019.

The summary is an abstractive summary written by an annotator with the marked mapping to the sentences from the original bug report.

Our annotation method was inspired by Rastkar *et al.* [1], where annotators assigned scores to each sentence if a sentence has to be included in summary set for OSCAR. For each sentence, the score is between zero and three. Zero ('0') means that it has not been linked by any annotator while '3' indicates that all three annotators have linked this sentence. A sentence is considered to be a part of the summary if it has a score of two or more. For each bug report, the set of sentences with a score of two or more (a positive sentence) is called the *Golden Summary Sentence*.[k] We adopted the same technique for annotating OSCAR corpus. However, only master bug reports were annotated in the OSCAR corpus.

Table 2 lists the total number of sentences in master bug reports for both OSCAR and MBRC corpora. It also lists the number of sentences selected by two or more annotators as well as by three annotators. Furthermore, it provides the average GSS values for both corpora.

Table 2. Details of the annotated subjects for the OSCAR and the MBRC corpora.

	OSCAR	MBRC
# of sentences	856	604
# of sentences selected by annotators	376	362
# of sentences selected by 2 or more annotators	171	190
# of sentences selected by 3 annotators	74	80
Average golden summary sentences	**19.98%**	**31.46%**
	Mean average	***25.72%***

4.3. *Features*

As done in the existing study [1], we extracted 24 conversational features from bug reports. These features were already applied and tested on bug reports by [1] and they reflected the corpora's indicators better than the text only. These features include the *location of words*, *frequency of words*, *sentence position*, *sentence conversion*, *time* and the *sentence similarly* characteristics. *Sentence Probability* represents the probability of a word appearing in one sentence while the *Turn Probability* represents the probability of a word on a conversational turn basis. Table 3 provides the short description of 24 features considered for our study. Full details on features are provided in [1].

4.4. *Research questions*

We examined the following research questions to explain and demonstrate our results.

[k]It is similar to Gold Summary Sentences as described by Rastkar *et al.* [1].

Table 3. Features key [1].

Feature ID	Description
SLEN	word count, globally normalized
SLEN2	word count, locally normalized
TPOS1	time from beginning of conversation to turn
TPOS2	time from turn to end of conversation
COS1	cos of conversation splits, w/*Sprob*
COS2	cos of conversation splits, w/*Tprob*
MXS	max *Sentence Probability* score
MNS	mean *Sentence Probability* score
SMS	sum *Sentence Probability* score
MXT	max *Turn Probability* score
MNT	mean *Turn Probability* score
SMT	sum *Turn Probability* score
CENT1	cosine of sentence & conversation, w/*Sprob*
CENT2	cosine of sentence & conversation, w/*Tprob*
TLOC	position in turn
CLOC	position in conversation
DOM	participants dominance in word
PENT	entropy of conversation up to sentence
SENT	entropy of conversation after the sentence
THISENT	entropy of current sentence
PPAU	time between current and prior turn
SPAU	time between current and next turn
BEGAUTH	is first participant (0/1)
CWS	rough *ClueWordScore* (cohesion of the conversation)

RQ1: Does Golden Summary Sentences in PRST ranked higher than the other sentences? To address this question, we need to rank the candidate sentences so that the top ranked sentences can be selected as the summary of a given bug report. More details are given in Sec. 5.1.

RQ2: How well does the PRST algorithm perform when applied to the MBRC corpus? We apply the proposed approach on MBRC corpus to investigate the performance of PRST algorithm. In addition, we discuss how α parameter influences PRST algorithm. Section 5.2 answers RQ2 in detail.

RQ3: What is the performance of the PRST algorithm on OSCAR corpus? We also apply the proposed approach on OSCAR corpus and discussed the influence of α parameter on PRST. The description that concerns how to implement this process and the evaluation results are shown in Sec. 5.3.

RQ4: What is the performance of PRST without duplicates? In our work, we have executed the evaluation experiment to compare the performance of PRST and PRST without duplicates. Moreover, we have discussed this experiment in details in Sec. 5.4.

In order to show the superiority of PRST, we organized research questions mentioned above. We verified if PRST truly worked for our scenario and how well it calculated the probabilities for generating summaries. In PRST, we adjusted the weighting factor α to achieve the best results. Meanwhile, we constructed two corpora, MBRC, and OSCAR, and calculated four statistical measures to evaluate the acquired results for our research.

4.5. *The BRC algorithm*

The BRC algorithm was based on the logistic regression model. Instead of generating an output of zero or one, it generated the probability of each sentence being part of the summary [1]. This algorithm was evaluated on bug reports by Rastkar *et al.* [1] earlier, based on the supervised learning that used 24 conversational features of a bug report. They used a linear classification from the *Liblinear Toolkit* to train the BRC algorithm on bug reports. In our research, we used the BRC algorithm as a benchmark for PRST.

4.6. *Evaluation measures*

To evaluate the effectiveness of both PRST and BRC algorithms, we adopted several statistical measures that compare summaries generated by algorithms to the golden summary sentences formed by the annotators. These measures are the four common Information Retrieval statistical metrics, namely, Precision, Recall, F-Score and Pyramid Precision (PP) [16]. They assessed the performance of algorithms against each other when tested on both MBRC and OSCAR bug report corpora.

For each bug report, the set of sentences with a score two or more is called the Golden Summary Sentence (*GSS*). We used *GSS* to measure the Precision and Recall. This technique is inspired by the early study proposed by Rastkar *et al.* [1]. Precision measures the percentage of sentences in a summary S that is also present in *GSS*. It is calculated as:

$$Precision = \frac{|S \cap GSS|}{|S|}.$$

Recall measures the percentage of sentences in GSS that are present in the summary being evaluated.

$$Recall = \frac{|S \cap GSS|}{|GSS|}.$$

As there is always a trade-off between Precision and Recall, being desirable but different features, the F-score is used as an overall measure. F-score combines the values of two other evaluation measures: Precision and Recall. F-Score can be computed by the following formula.

$$F\text{-}score = 2 * \frac{precision * recall}{precision + recall}.$$

We also used Pyramid Precision, which is a normalized evaluation measure taking into account the multiple annotations available for each bug report [1]. The larger the values are, the better the results will be.

$$PP = \frac{\#\ annotation\ times\ of\ top\ ranked\ sentences}{max\ times\ a\ sentence\ is\ annotated}.$$

We use the four metrics described above to compare and evaluate the results of our PRST algorithm on both MBRC and OSCAR corpora.

4.7. *Evaluation method*

In this case study, we employed leave-one-out-cross-validation procedure to determine the training and testing sets. One round of leave-one-out-cross-validation involves partitioning a sample of data into subsets, validating the analysis on one subset — called the testing set, and performing the analysis on the other subsets — called the training set. To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

5. Results

Figure 5 shows an example of a generated extractive summary from the OSCAR corpus. In the following subsections, we discuss the results of each RQ in details.

5.1. *RQ*1: *Top ranked sentences*

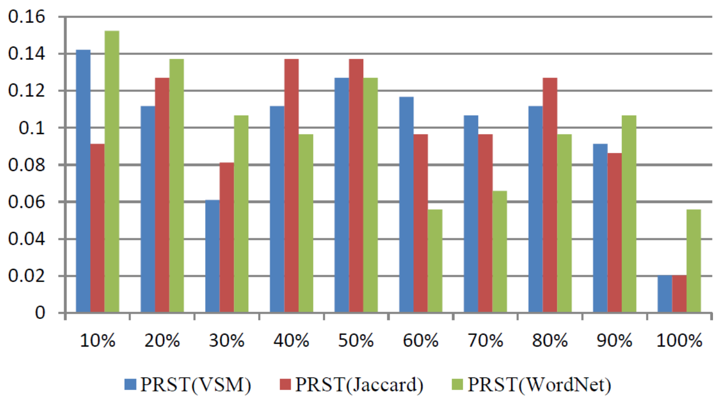
We setup this RQ to check whether the summary sentences belonged to the top ranking sentences. In our study, we distributed sentences into a threshold of 10% sentences each and examined the distribution of GSS. As described, we divided all sentences into 10 portions where each portion comprised of 10% of sentences. For every portion, we calculated the percentages of sentences belonging to GSS. If it showed a downward trend, this implied that the GSS for sentences ranked higher that the non-GSS sentences. In such a way, we measured the performance of PRST.

Figure 6 shows the histogram depicting the ground truth values for three variants of the PRST. Where x-axis depicts the 10% distribution increase or threshold value for all variants of the PRST while the y-axis shows the GSS. Among first 10% sentences, GSS of Jaccard is less than the GSS of VSM and WordNet. If the threshold (T) is between 10% to 20%, the GSS of WordNet is still higher than the VSM and Jaccard. The GSS of VSM has a swift decline in the 20% to 30% sentences, which only reaches 0.06, meanwhile, the one of Jaccard is 0.08. However, with the increase of threshold value from 30% to 40% and above, the Jaccard and the VSM performed better than the WordNet except for $T = 90\%$ and 100%.

Compared with Fig. 6(a) for MBRC corpus, the trends of ground truth values in Fig. 6(b) for OSCAR corpus were clear. The GSS of WordNet peaks at about 0.16 among the 10% sentences and it tends to decrease gradually onwards. Among the 100% sentences, it finally reduces to 0.04. The GSS of both VSM and Jaccard peak at

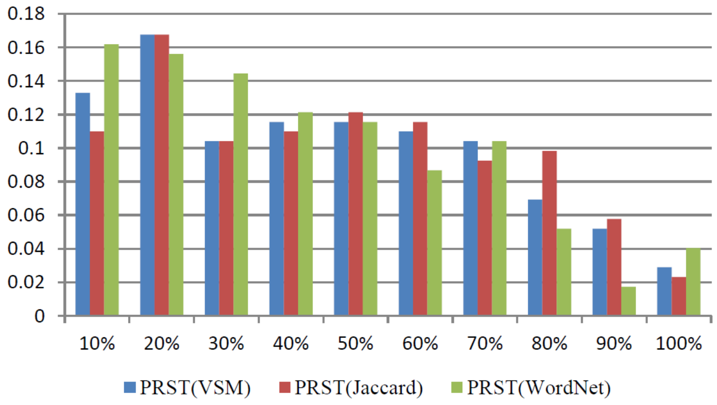
```
Bug id: 128682
Title: dragging linked images causes drop of relative image
    path, not absolute link target.

Duplicate bug: 128416
Duplicate bug title:  dragging a image link to a tab loads
    the image not the link.

Summary: open http://www.mozilla.org in one window. drag the
     logo banner at the top (a link to http://www.mozilla.
    org) into another window. the browser will try to load
    http://www.images.com/mozilla-banner.gif. i'm seeing it
    on 0301 linux, and daa confirmed it on 0302 win32. this
    is a 0.9.9-critical issue, imo.findparentnode(
    draggednode, ns_literal_string(``a").get(),
    getter_addrefs(linknode)). don't we want to do
    getanchorurl(linknode, urlstring). use linknode to fill
    urlstring if we're dragging image-within-link this works
     for me, and seems ``obviously right". soliciting quick
    reviews and approval for 0.9.9 and 1.0. *** bug 128416
    has been marked as a duplicate of this bug ***.
```

Fig. 5. Extracted summary for the bug # 128682 taken from the Mozilla Bug Repository. This bug report is a part of OSCAR corpus.



(a) MBRC

Fig. 6. (Color online) GSS values for MBRC and OSCAR corpora.



(b) OSCAR

Fig. 6. (*Continued*)

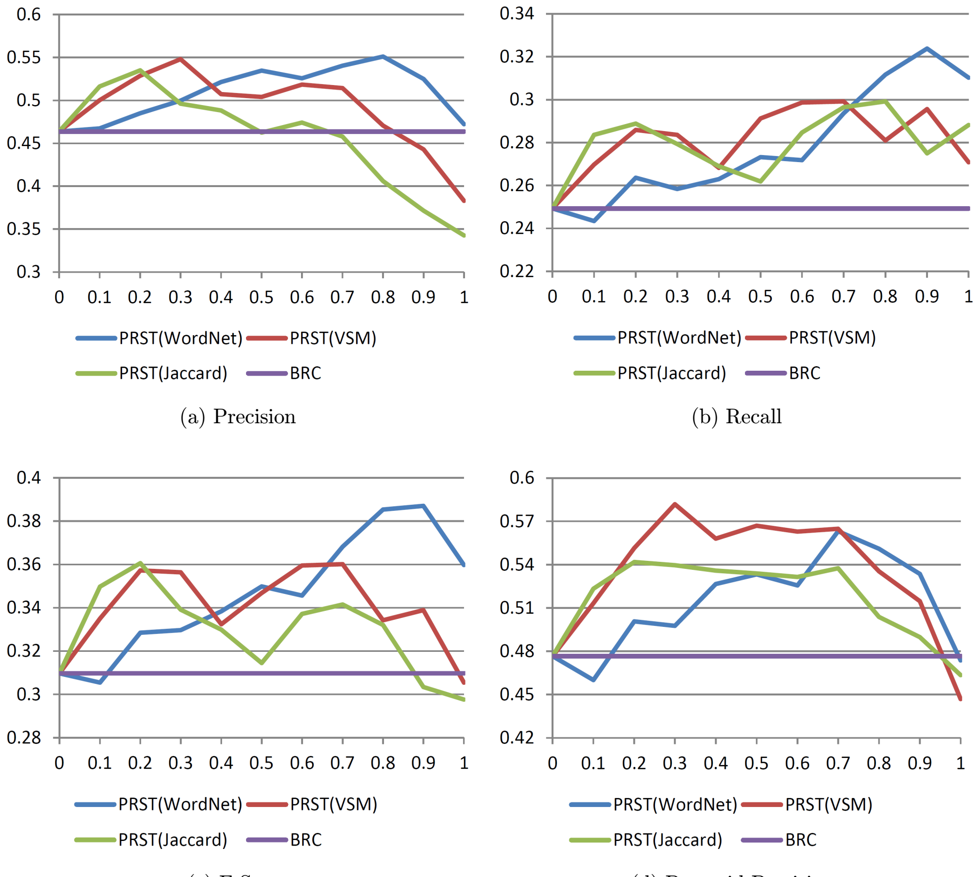
about 0.16 among the 20% sentences; however, from 30% to 100% sentences, the GSS of VSM and Jaccard fluctuate slightly.

In short, for 10% sentence distribution, the downward trend is more apparent in the OSCAR corpus than that in the MBRC. By applying different α values, a weighting factor, on the corpora we achieved different results. On verifying all resulted values we found that $\alpha = 0.82$ for MBRC and $\alpha = 0.78$ for OSCAR gave best results for the PRST algorithm.

5.2. *RQ2: Performance of PRST on MBRC corpus*

By adjusting the weighting factor α, we ran the PRST algorithm on MBRC and obtained values for four statistical measures, Precision, Recall, F-score, and Pyramid Precision. Figure 7 shows the Precision, Recall, F-score, and Pyramid Precision values, ranging between 0 to 1, for all three variants of the PRST, and the original BRC algorithm when applied to the MBRC corpus.

The line graphs compared the trends of the evaluation matrices, Precision, Recall, F-Score, and Pyramid Precision (PP) when applied on three variants of PRST and a benchmarking algorithm the BRC on the MBRC corpus and compared values. In line graphs ‘a’, ‘b’, and ‘c’, the PRST(WordNet) outperformed other variants and the BRC algorithm. However, in graph ‘d’, the PRST(VSM) outperformed other variants of PRST as well as the original BRC algorithm. Hence, for the MBRC corpus, the PRST with VSM worked the best with the PP $> 56\%$, whereas, BRC algorithm performed the worst with PP $< 48\%$. Except for the PP, the PRST based on WordNet showed the best results for remaining matrices. For instance, when the parameter α reached 0.82, Precision of the PRST (WordNet) equaled to 55.78%, whereas the Precision for the BRC equaled 46.31%. Therefore, based on these observations, we could clearly infer that the PRST with WordNet outperformed the BRC by 9.47% on the basis of the Precision.



(a) Precision (b) Recall (c) F-Score (d) Pyramid Precision

Fig. 7. (Color online) Precision, Recall, F-Score and Pyramid Precision of the PRST and the BRC classifiers on the MBRC corpus.

Table 4 provides the best results for Precision, Recall, F-Score, Pyramid Precision scores as well as the running time for BRC algorithm and all three variants of the PRST algorithm. Each value of weighting factor α is examined to achieve best results.

5.3. *RQ3: Performance of PRST on OSCAR corpus*

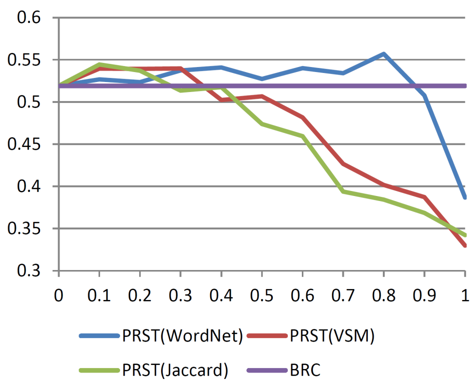
Same as in RQ2, we applied all four statistical measures to evaluate the performance of the PRST on an OSCAR corpus. Figure 8 shows the detailed results and Table 5 shows the best results of Precision, Recall, F-score and Pyramid Precision when using all three variants of the PRST and the BRC algorithms on an OSCAR corpus.

The line graphs (in Fig. 8) compared the trends of the evaluation matrices when applied on three variants of the PRST and the benchmarking algorithm BRC on an OSCAR corpus. In all four graphs, PRST with WordNet performed far better than

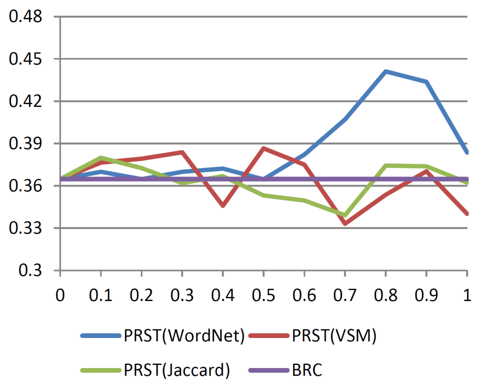
Table 4. Comparison of the BRC and the PRST algorithms on the MBRC corpus.

Algorithm	α	Precision (%)	Recall (%)	F-score (%)	PP (%)	Running time (s)
BRC		46.31	24.93	30.97	47.67	64
PRST (VSM)	0.26	54.83	30.28	37.18	**56.84**	4218
PRST (Jaccard)	0.22	54.08	29.42	36.62	54.58	4202
PRST (WordNet)	**0.82**	**55.78**	**32.80**	**39.89**	**55.95**	**4316**

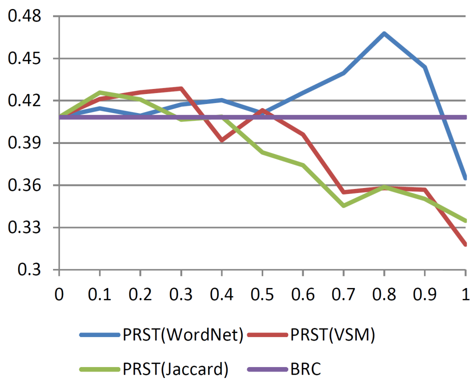
other variants, as well as the existing BRC algorithm. The precision of the PRST with WordNet reached 56.67%, when the α parameter was set to 0.78, which indicated the best performance. Similar to the MBRC, the BRC algorithm performed the worst with the precision of 51.9%. Thus, the PRST on an OSCAR outperformed the BRC by 4.7%.



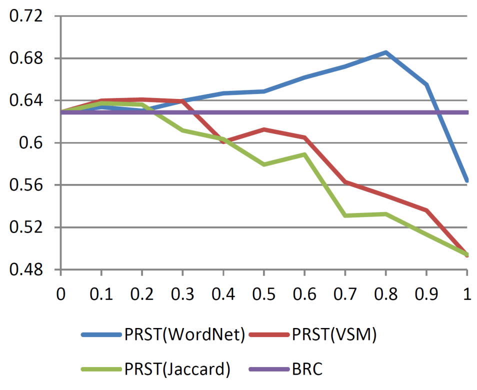
(a) Precision



(b) Recall



(c) F-Score



(d) Pyramid Precision

Fig. 8. (Color online) Precision, Recall, F-Score and Pyramid Precision for the PRST and the BRC classifiers on an OSCAR corpus.

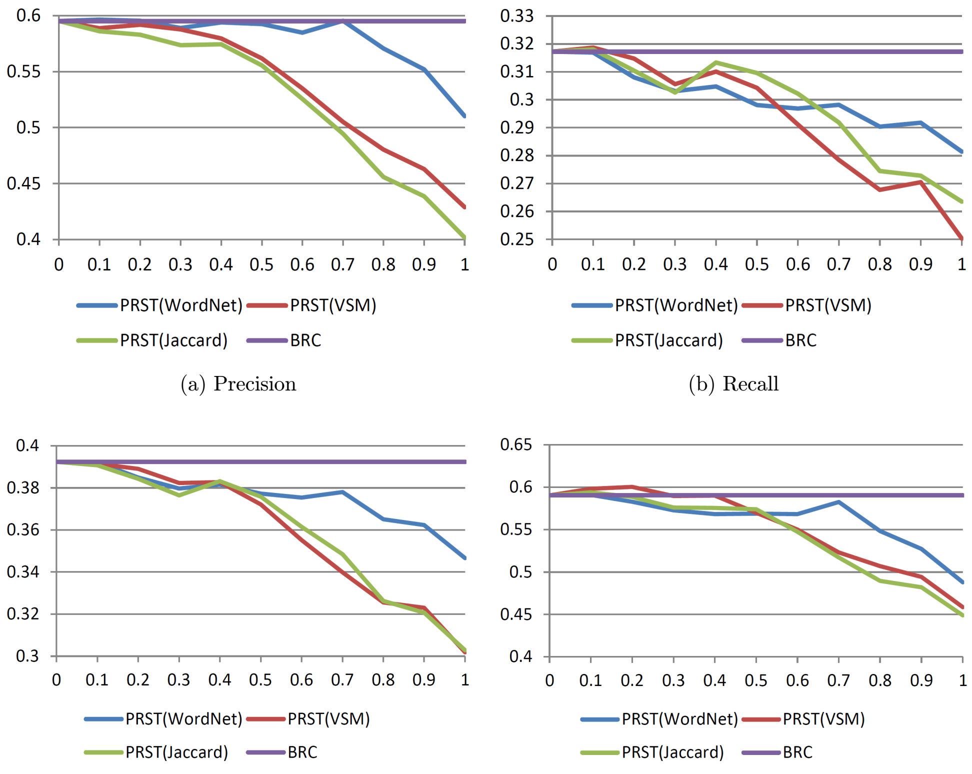
Table 5. Comparison of BRC and PRST algorithms on OSCAR corpus.

Algorithm	α	Precision (%)	Recall (%)	F-score (%)	PP (%)	Running time (s)
BRC		51.91	36.47	40.83	62.87	66
PRST (VSM)	0.24	55.15	38.86	42.54	65.10	8691
PRST (Jaccard)	0.10	54.44	37.99	42.58	63.74	8718
PRST (WordNet)	**0.78**	**56.67**	**44.29**	**47.34**	**68.93**	**8847**

As seen in Table 5, the running time of the PRST algorithm is longer than the BRC. It is due to the fact that PRST includes the training time as well. The testing or the prediction time is almost the same for both PRST and the BRC algorithms.

5.4. *RQ4: Performance analysis of PRST on BRC corpus*

In RQ4, we analyze the performance of PRST algorithm by comparing it with BRC algorithm on the original BRC corpus. Figure 9 and Table 6 illustrate that the PRST performance declines when using the original BRC corpus (without duplicates).



(a) Precision

(b) Recall

(c) F-Score

(d) Pyramid Precision

Fig. 9. (Color online) Precision, Recall, F-Score and Pyramid Precision of PRST and BRC classifiers on original BRC corpus without duplicates.

Table 6. Performance comparison of BRC and PRST algorithms on the original BRC corpus.

Algorithm	α	Precision (%)	Recall (%)	F-score (%)	PP (%)	Running time (s)
BRC		59.51	31.72	39.23	59.04	83
PRST (VSM)	0.02	59.64	32.07	39.51	59.71	2689
PRST (Jaccard)	0.06	59.62	**32.31**	**39.70**	**60.39**	2770
PRST (WordNet)	0.22	**59.92**	31.11	38.84	58.04	2886

In Fig. 9, the data on X-axis indicate the values of the weighting factor α while the data on Y-axis show the values of respective statistical measures i.e. Precision, Recall, F-Score, and Pyramid Precision. Varying the weighting factor α generates the different evaluation results. The best results are shown in Table 6.

The performance of the PRST decreased when tested on the original BRC as it lacked duplicate bug reports. For instance, the F-Score value for PRST (WordNet) is 47.24% when α is 0.22 on MBRC corpus. Whereas, in original BRC it is decreased to 38.48% with the same α value. Similarly, the F-Score for three variants of the PRST declines with the change in α when compared to that of BRC in Fig. 8. Moreover, comparing the results shown in Table 5 as well as the Figs. 6 and 7 depicts that our PRST techniques performed far better than the existing BRC when duplicate bug reports were involved. Therefore, our hypothesis of utilizing the textual information in duplicate bug reports for summarizing master bug reports has proved to be a promising one. Moreover, it has performed substantially better than the BRC algorithm.

5.5. *Threats to validity*

In this section, we identify internal and external threats to the validity of our study.

5.5.1. *Internal threats*

We have found two major internal threats in our study, which are *sentence division* and *annotation of corpora.*

Sentence Division: One of the primary threats to the internal validity is slicing paragraphs into sentences using the *LingPipe Toolkit.* It is a powerful tool, which can divide sentences into words, identify sentence boundaries, and perform other tasks, such as speech and word analysis. While writing bug reports, the reporters use a wide variety of sentence patterns and punctuation transformations. It is possible that the LingPipe Toolkit may not provide accurate results for some sentence patterns since different developers have different writing habits. In order to minimize this risk we used LingPipe Toolkit for cropping paragraphs into sentences only.

Annotation: Another threat is about the personal behaviors of annotators. Annotators have a different understanding of bug reports and can interpret bug reports differently as each bug report has a different structure. To minimize this risk, we choose three annotators for bug reports annotation. If two participants agree on a sentence to be a part of the summary, that sentence is included in the summary. By considering the results of three annotators, we generate the final summary of master bug reports.

5.5.2. *External threats*

Availability of Corpus: As discussed in Sec. 1, our main idea is to employ duplicate bug knowledge for summarizing master bug reports. While selecting an appropriate corpus for experimentation, we noticed that there are very few publicly available corpora that contain master and duplicate bug reports mapped together. In addition, it was difficult for us to apply appropriate methods on publicly available corpora as they lack duplicate bug reports, to form a better summary. Therefore, we constructed a new corpus i.e. OSCAR. We plan to construct a new larger corpus and implement our technique to other comparatively larger publicly available corpora in the future to generalize our results.

6. Related Work

Since 1958 [17], the automated summarization of natural language text has been widely studied. This was followed by many papers in different directions to improve text summarization. These techniques varied from simple *tf-idf* based techniques to more complex machine learning-based methods [2]. Generally, two basic approaches have been employed to generate summaries i.e. *extractive* and *abstractive*. These approaches are further broadly categorized into Supervised and Unsupervised machine learning methods.

6.1. ***Summarizing bug repositories***

Murray and Carenini [18] developed a summarizer for conversations like meetings and emails, and Rastkar *et al.* [1] further employed the same extractive supervised learning approach for summarizing bug repositories. As bug reports resemble conversations, their approach created an extractive summary, which selected a set of sentences from the original bug report to compose an informative and organized summary. The approach used a logistic regression classifier trained on a corpus of manually created reference bug report summaries called *golden summary*.

In contrast, Mani *et al.* [2] applied unsupervised approaches on the same corpus of bug reports, dealing with noise in bug reports by introducing a noise reducer for generating better summaries. Similarly, Lotufo *et al.* [19] proposed a PageRank-based unsupervised approach for bug report summarization considering the importance of evaluation links, title, and description similarity.

Our Approach: Our supervised approach, though using the same logistic regression classification as in [1], differed in the sense that we compared bug reports with their duplicates as duplicate bug provide additional information. Furthermore, we developed three variants of PageRank algorithm, which we called the **PRST**. It utilized VSM, Jaccard, and WordNet models for measuring the similarity between bug report sentences. In addition, we reconstructed the original BRC corpus to accommodate duplicate bug reports as well as constructed a new corpus, OSCAR, to extract better summaries, thus generalizing our results. Empirical results illustrated that the PRST outperformed the state-of-the-art method BRC in terms of Precision, Recall, F-score, and Pyramid Precision. Meanwhile, the PRST with WordNet achieved best results against the PRST with VSM and Jaccard. Table 7 provides a summary of related work in bug report summarization. It also gives an overview of our approach and compares it with existing bug report summarization approaches.

Table 7. Summary of related work on bug report summarization.

Previous work	Approach	Model	Features
[1] 2010	Supervised	Logistic Regression	Bug report conversations
[2] 2012	Unsupervised	MMR + Grasshopper + Centroid + DivRank	Noise Reducer
[19] 2012	Unsupervised	Markov Chain + PageRank	Bug report title & description + frequently discussed topics + sentence relevance
Our approach	Supervised	PageRank + VSM + Jaccard + WordNet	Textual similarity between master and duplicate bug reports

6.2. *Summarizing source code repositories*

For source code repositories, Haiduc *et al.* [20] generated term-based summaries for methods and classes that contained the set of most relevant terms to define code entities. Similarly, Sridhara and colleagues [21] defined a technique to generate natural language summary comments for an arbitrary Java method by manipulating both structural and natural language evidences in the method based on heuristics. In another research, Rastkar *et al.* [22] used automated documentation generation approach and a task-based evaluation for generating light abstractive summaries for a crosscutting code for concern.

Ying *et al.* [23] first proposed a supervised learning approach for summarizing code fragments, focusing on presentation aspect of code examples. As defined by [23] code fragments are partial programs that serve the purpose of demonstrating the usage of an API. Any line in the summary is more important in the context of a query and a syntax than any other line in a code fragment. As an initial investigation, they exploited syntactic and query features of a code fragment by applying machine learning techniques to train the classifier and attained the precision of 71%. Recently, Nazar *et al.* [24] generated source to source summaries of code fragments by utilizing crowd enlistment on a smaller scale with supervised machine learning algorithms. They found that their approach produced statistically better summaries than the existing study [23] with 82% Precision. Other efforts focused on recommending code snippets [25, 26] and finding tutorial segments explaining APIs [27].

7. Conclusion

In this paper, we have proposed a PageRank-based Summarization Technique (PRST) which effectively utilized the textual information of duplicate bug reports for generating extractive summaries of master bug reports. We applied three variants of PageRank algorithm based on VSM, Jaccard, and WordNet similarity metrics, to measure important sentences in master and duplicate bug reports. Additionally, we trained a regression model, using supervised learning, to predict the probability values of every sentence belonging to the summary of the master bug report. Finally, we combined the values of PageRank and regression model scores to rank the sentences and selected top 25% sentences as summary sentences. By evaluating the results on two bug reports corpora namely MBRC and OSCAR, we found that our PRST algorithm outperformed the BRC algorithm in terms of Precision, Recall, F-Score, and Pyramid Precision evaluation measures. Meanwhile, the PRST based on WordNet can generate a better summary of the master bug reports than other variants of the PRST.

In the future, we plan to investigate whether other attributes of bug repository such as dependent bugs can be employed for summarizing bug reports. We shall also consider using our approach in an unsupervised manner to further evaluate the usefulness of duplicate bug reports in summarizing master bug reports.

Acknowledgments

We would like to extend our gratitude to the individuals who have dedicated their time and effort in performing annotation of our bug report corpus. We would also like to thank all anonymous reviewers for their valuable comments. This work is supported in parts by the National Basic Research 973 Program of China under Grant No. 2013CB035906, the Fundamental Research Funds for the Central Universities of China under Grant No. DUT13RC(3)53, and in part by the New Century Excellent Talents in University of China under Grant No. NCET-13-0073 and the Natural National Science Foundation of China under Grant No. 61370144.

References

1. S. Rastkar, G. C. Murphy and G. Murray, Summarizing software artifacts: A case study of bug reports, in *Proc. IEEE Int. Conf. Software Engineering*, 2010, pp. 505–514.
2. S. Mani, R. Catherine, V. S. Sinha and A. Dubey, Ausum: Approach for unsupervised bug report summarization, in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering*, 2012, pp. 1–11.

3. S. Rastkar, G. C. Murphy and G. Murray, Automatic summarization of bug reports, *IEEE Trans. Softw. Eng.* **40**(4) (2014) 366–380.
4. Princeton University, Wordnet: A lexical database for English, 2014, http://wordnet.princeton.edu/.
5. C. D. Manning, P. Raghavan and H. Schütze, *Introduction to Information Retrieval* (Cambridge University Press, Cambridge, 2008).
6. P. Tan, M. Steinbach and V. Kumar, *Introduction to Data Mining (First Edition)* (Addison-Wesley Longman, Boston, 2005).
7. N. Jalbert and W. Weimer, Automated duplicate detection for bug tracking systems, in *Proc. 38th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks*, 2008, pp. 52–61.
8. N. Bettenburg, R. Premraj, T. Zimmermann and S. Kim, Duplicate bug reports considered harmful...really?, in *Proc. 24th IEEE Int. Conf. Software Maintenance*, 2008, pp. 337–345.
9. J. Xuan, H. Jiang, Z. Ren and W. Zou, Developer prioritization in bug repositories, in *Proc. 34th Int. Conf. Software Engineering*, 2012, pp. 25–35.
10. J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo and X. Wu, Towards effective bug triage with software data reduction techniques, *IEEE Trans. Knowl. Data Eng.* **27**(2015) 264–280.
11. X. Wu, V. Kumar, R. J. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. Yu, Z. Zhou, M. Steinbach, D. J. Hand and D. Steinberg, Top 10 algorithms in data mining, *Knowl. Inf. Syst.* **14**(1) (2008) 1–37.
12. W. Li, M. Wu, Q. Lu, W. Xu and C. Yuan, Extractive summarization using inter- and intra- event relevance, in *Proc. 21st Int. Conf. Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 2006, pp. 369–376.
13. G. Salton, A. Wong and C. S. Yang, A vector space model for automatic indexing, *Commun. ACM* **18**(11) (1975) 613–620.
14. G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross and K. Miller, Wordnet: An on-line lexical database, *Int. J. Lexicogr.* **3** (1991) 235–244.
15. 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. Software Engineering*, 2012, pp. 14–24.
16. T. Prifti, S. Banerjee and B. Cukic, Detecting bug duplicate reports through local references, in *Proc. 7th Int. Conf. Predictive Models in Software Engineering*, 2011, pp. 1–9.
17. H. P. Luhn, The automatic creation of literature abstracts, *IBM J. Res. Dev.* **2**(2) (1958) 159–165.
18. G. Murray and G. Carenini, Summarizing spoken and written conversations, in *Proc. Conf. Empirical Methods in Natural Language Processing*, 2008, pp. 773–782.
19. R. Lotufo, Z. Malik and K. Czarnecki, Modelling the 'hurried' bug report reading process to summarize bug reports, in *Proc. 28th IEEE Int. Conf. Software Maintenance*, 2012, pp. 430–439.
20. S. Haiduc, J. Aponte, L. Moreno and A. Marcus, On the use of automated text summarization techniques for summarizing source code, in *Proc. 17th Working Conf. Reverse Engineering*, 2010, pp. 35–44.
21. G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock and K. Vijay-Shanker, Towards automatically generating summary comments for java methods, in *Proc. 25th IEEE/ACM Int. Conf. Automated Software Engineering*, 2010, pp. 43–52.
22. S. Rastkar, G. C. Murphy and A. W. J. Bradley, Generating natural language summaries for crosscutting source code concerns, in *Proc. IEEE 27th Int. Conf. Software Maintenance*, 2011, pp. 103–112.

23. A. T. T. Ying and M. P. Robillard, Code fragment summarization, in *Proc. Joint Meeting of the European Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering*, 2013, pp. 655–658.
24. N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li and Z. Ren, Source code fragment summarization with small-scale crowdsourcing based features, *Front. Comput. Sci.* **10**(3) (2016) 504–517.
25. H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang and X. Luo, Rosf: Leveraging information retrieval and supervised learning for recommending code snippets, *IEEE Trans. Serv. Comput.* **PP**(99) (2016) 1–1.
26. L. Nie, H. Jiang, Z. Ren, Z. Sun and X. Li, Query expansion based on crowd knowledge for code search, *IEEE Trans. Serv. Comput.* **9**(5) (2016) 771–783.
27. H. Jiang, J. Zhang, X. Li, Z. Ren and D. Lo, A more accurate model for finding tutorial segments explaining APIs, in *23rd Int. Conf. Software Analysis, Evolution, and Reengineering*, 2016, pp. 157–167.