


text of this document as if you were reading it naturally.
Results indicate that VSM+SA and VSM+TR both outperformed the baseline VSM, but found that VSM+SA+TR was the best performing approach overall. In the user study, we evaluated best performing approach (i.e., VSM+SA+TR) to the baseline VSM. Our choice for using VSM as a baseline is supported by four premises: i) VSM can determine term relevance in bug report queries [49], ii) each discussion comment can be represented into a vector of term-frequency-inverse-document frequency (TF×IDF) weights that can be enhanced for relevance ranking [44], iii) VSM can be enhanced by NLP methods when VSM alone fails to capture textual matching between queries and comments [67], and iv) it is regarded as a suitable method for bug localization tasks [60, 65, 66]. Results indicate that RETRORANK when using combination VSM+SA+TR outperformed the baseline VSM.

II. Problem and Scope

One of the challenges directly affecting developer’s productivity in the process of fixing bug reports is the time required to search and evaluate high volumes of reported bugs. Large software systems are plagued by thousands of bug reports daily [8]. As this number keeps increasing, the time needed to investigate these bug reports by developers becomes almost an unattainable task to achieve [33]. Additionally, some of the bug reports are more complex than others. They contain discussion threads that are lengthy and frequently become difficult to understand as different stakeholders engage in extensive discussions [5]. Lengthy discussions pose substantial challenge to developers who must reason about which comments in a discussion thread are relevant to finding a potential solution for the bug report that they are investigating.

To automate this process, our tool effectively and efficiently recommends the most relevant bug-fixing comments. To demonstrate the output of our tool, we provide an example. In Figure 1a, we show an unresolved LibreOffice bug with ID #34600. In Figure 1b, we show a collection of related bug-fixing comments recommended from lengthy discussion threads from various resolved past bugs. These recommended comments fall under the common theme of bug fix #34600. For example, ‘comment 3’ extracted from discussions of bug #34436 provides concrete suggestion on what condition of bug #34600. Similarly, ‘comment 2’ extracted from discussions of bug #33622 provides directions on how to handle the 90 degree alignment issue. While ‘comment 4’ extracted from discussions of bug #34136 and ‘comment 2’ of bug #32795, confirm the fix of bug #33622.

Note that by using plain keyword-matching heuristics for extracting comments with keywords ‘fix’ or ‘fixed’ would be an insufficient search criterion, as there exist other insightful comments that could capture sentiment features and rich patterns that are more affirmative [7]. Such insightful information is present in the comments from bugs #33622 and #34436, both characterized by positive sentiment and semantic relevance. This sentiment pattern was detected by using Sentiment Analysis. The semantic connection between descriptions of comments was obtained by using TextRank.

A. Vector Space Model

Vector Space Model (VSM) is a similarity-based ranking model [57]. This model takes each comment and a user query and represents it as an individual document vector \( \mathbf{d} \) of term weights. These weights are determined using term frequency-inverse document frequency TF×IDF [55]. This gives a vector \( \hat{d} = (w_{1,d}, w_{2,d}, ..., w_{n,d}) \), where \( w_{i,d} \) is the term weight for the \( i \)th term in document \( d \). Once the weighted vectors are defined, the cosine similarity metric computes a similarity score. The cosine similarity between two documents \( d_1 \) (i.e., comments) and \( d_2 \) (i.e., user query) is computed as follows:

\[
sim(d_1, d_2) = \frac{\sum_{i=1}^{n} w_{i,d_1} w_{i,d_2}}{\sqrt{\sum_{i=1}^{n} w_{i,d_1}^2} \sqrt{\sum_{i=1}^{n} w_{i,d_2}^2}}
\]

The expression in (Eq.1) generates a similarity score between each comment and a user query input. That means, the higher the ranking score of each comment is, the more likely the user query input is associated with the given comment.
B. TextRank Model

TextRank (TR) is a graph-based ranking model [56]. This model aims to identify relations between various entities in a text by incorporating the concept of text summarization [23, 31, 35, 37, 48, 56]. The task of text summarization is described in context of finding the relevant sentences in a text and extracting salient terms. The input to the model is a connected graph \( G = (V, E) \) with the set of vertices \( V \) and set of edges \( E \), where \( E \) is a subset of \( V \times V \). The output of the model is a list of top-N terms with corresponding ranked scores extracted from a corpus. Vertices denote each candidate term. Edges between terms are added between two candidate terms based on their co-occurrence relation. The co-occurrence relation indicates the relationships between terms. In our context, we adapted the TextRank graph with undirected edges for each bug comment. An edge is added between two candidate terms in a comment. Then, the weight of the edge is computed where the two words co-occur relative to the number of comments. The TextRank score \( TR(V_i) \) associated with a candidate term at vertex \( V_i \) is iteratively updated as follows:

\[
TR(V_i) = 1 - d + d \sum_{V_n \in R(V_i)} \frac{w_{ni}}{\sum_{V_m \in E(V_i)} w_{nm}} TR(V_n)
\]

where \( w_{ni} \) is the strength of the connection between two vertices \( V_n \) and \( V_i \), \( E(V_i) \), and \( d \) is the damping factor commonly set to 0.85 [36, 43]. \( TR(V_n) \) reports the importance score assigned to a term. The algorithm initially initializes each term with score of 1. The rank update continues iterating until convergence. The convergence usually ends after 20 – 30 iterations. The terms are then sorted based on ranking score. Finally, a list of terms with top-N ranked scores are extracted.

C. Sentiment Analysis

Sentiment Analysis (SA) which is sometimes referred to as Opinion Mining, is a subarea of NLP [14, 22, 40, 46, 61]. SA studies people’s opinions, sentiments, and attitudes toward entities in a text. SA aims to classify words into two categories: ‘good’ and ‘bad’, and then measure an overall good/bad score in a text. SA can be described as a classification process. The goal of SA is to achieve the following tasks: i) find opinions, ii) identify the sentiments they express, and iii) classify their sentiment. Conceptually, this process is designed in four steps. The first step in SA is the Sentiment Identification. The input is a text from any domain. The goal is to determine whether the content of the text is subjective or objective. The second step in SA process is Feature-Based Classification. The feature-based classification task treats the text either as bag-of-words (BOWs), or as a string which retains the sequence of words in the document. The third step in SA process is the task of Sentiment Classification, which includes machine learning classification approach or lexicon-based approach. Either of these techniques can be applied to one of the three classification levels: document-level, sentence-level, or aspect-level. In our approach, we employ lexicon-based classification applied to document-level (i.e., comments). We decided on following this approach because it involves calculating orientation for a bug-fixing comment from the semantic orientation of words or phrases in the comment. Hence, lexicon-based classification relies on a sentiment lexicon, a collection of known and precompiled sentiment terms. It depends on finding the opinion lexicon by using corpus-based approach. The corpus-based approach begins with a seed list of opinion words, and then finds other opinion words in a large corpus to help in finding opinion words with context specific orientation. The fourth step in SA process is the Sentiment Polarity. The sentiment polarity of a word can be identified by computing the occurrence frequency of the word in a large annotated corpus of texts [51]. If the word occurs more frequently among positive texts, then its polarity is positive. If it occurs more frequently among negative texts, then its polarity is negative. If it has equal frequencies, then it is a neutral word. Our focus is to find the sentiments polarity [21] of terms in bug comments as described in the forth step. To achieve this goal, we used NLP python module TextBlob Sentiment Classifier [28] for processing text. TextBlob takes the text and returns a blob with sentences splitted. Then, built-in sentiment analyzer performs the following functionalities such as extracting terms using Naive Bayes [34] and calculating sentiment for each term using a lexicon-based score. The lexicon-based score is obtained by using averaging technique [15, 63], in which case each term returns sentiment polarity score. The polarity score is determined within ranges of \([-1.0, 1.0]\), where 1.0 is highly positive and -1.0 is highly negative. An example to a positive comment is: “Try using String class […] it works fine […]”.

IV. Approach

Our ranked-based approach recommends (RetroRank) a list of comments by leveraging a combined weighted function where keyword weights of each comment are influenced by scores obtained by Vector Space Model (VSM), Sentiment Analysis (SA), and the TextRank Model (TR).

A. Overview of Architecture

The underlying architecture of RetroRank is depicted in Figure 2. The key components are the local database containing Bug Repositories (i.e., Bugzilla), the Comments with Keyword Scores, the Sentiment Analysis that computes sentiment keyword scores based on lookup bonus/penalty opinion lexicon, and TextRank that ranks keywords based on their co-occurrence and lexical order of importance (i.e., keyword that is most similar to all the other keywords found in comments). RetroRank is structured into four blocks each representing a set of components. The blocks include: i) Data Collection and Preprocessing, ii) Keyword Extraction and Computed Scores, iii) Adjusted Keyword Scores, and iv) Retrieved Comments. The details are described as follows.

- **Data Collection and Preprocessing:** refers to the block represented by the set of components (labels 16, 23 and 48). Figure 2. The input to the block are Bug Repositories extracted from their XML schema. First, component Metadata XML Parser 16 parses the metadata. The extracted metadata are: i) Pre-defined attributes of the bug and ii) Free-form text
that includes bug comments. Second component, Local Bug Database saves the metadata that are needed to establish relationships between the bug reports and discussion comments. Third component, pre-processes comments into four consecutive steps: i) Tokenization, ii) Normalization, iii) Stop-Word Removal, and iv) Stemming. Tokenization tokenizes text into terms. Normalization aims to eliminate punctuation, non alphanumeric symbols, converting terms to lowercase. Stop-Words Removal removes words that carry little meaning. We use a standard NLTK library to perform this task. Stemming reduces terms to its root form. For example, terms 'crashing' and 'crashed' reduce to their base term 'crash'. We use the well-known Porter algorithm to perform this step. Same pre-processing steps also apply to component that represents terms formulated as user query inputs.

**Keyword Extraction and Comment Computed Scores:** refers to the block represented by the set of components (labels 3). Initially, we perform the task of keyword extraction. Keyword extraction selects the keywords that best describe a comment. In our approach, ‘Comments with Initial Keyword Scores’ (label 3) is the component where we extract keywords and compute initial keyword weights for each comment and a user query using the TF×IDF schema. In our approach, TF×IDF keywords are words in comments, documents are comments in the bug reports, and the set of documents is the set of all comments in all bug reports. Then we apply the VSM (see Section III-A Eq.1) to compute the similarity score between each comment and a user query (label 3). Next, we construct the keyword dictionary for Sentiment Analysis Dictionaries component (label 3). We incorporate the sentiment analysis (SA) process to keyword extraction as described in Section III-C. The process to construct the SA dictionary was based in three steps: i) filter comments across all bugs that are in: RESOLVED, VERIFIED, or CLOSED status. ii) represent each comment as a graph in which each word is a vertex and each edge is the strength between a pair of vertices. When one vertex links to another, it casts a vote for that vertex. The higher the number of votes that are cast to a vertex, the higher the importance of that vertex, iii) iterate through graph until it convergences using (Eq.2), and iv) finally, sort the vertices based on their final scores and use the values attached to each vertex for ranking.

**Adjusted Keyword Scores:** refers to the block represented by the set of components (labels: 3). The initial scores from VSM (label 3) are then adjusted and boosted by scores of keywords in SA dictionaries (label 3), Figure 2, and by scores of keywords in TR dictionaries (label 3, Figure 2). This combined boosting weighted function $F_w = f(VSM) = SA + TR$ re-ranks each comment’s initial VSM keyword scores to more favorable positions in the list influenced by both SA and TR scores. For example, the actual computed final scores for each comment are shown in our tool in Figure 3 column 7 denoted as ‘Comment Score’.

**Retrieved Comments:** refers to the block represented by the set of components (labels: 3). Figure 2. Given a User Query, (label 3), Figure 2, we filter comments with adjusted keyword scores from the previous step in descending order, and iii) finally, we retrieve the top 10 highest Recommended Comments to the user (label 3, Figure 2).
In practice, RETRO RANK is designed to adapt to any commercial or OSS project that support modern development practices. The inclusion of 'pre-merge' code review [4, 39] is an example where relevant comments can be used as part of code review [20, 38]. Currently, RETRO RANK supports four OSS projects (i.e., GCC Compiler, Eclipse IDE, LibreOffice, and Apache Server). A screenshot in Figure 3 shows a list of recommended bug-fixing comments that can be used to address bug #61852. In this example, the RETRO RANK recommended several past fixed bugs (i.e., #26494, #40989, #53063, and #52304) and their associated bug-fixing comments shown in column 8;

V. EVALUATION STRATEGY

Our evaluation strategy is to conduct a laboratory synthetic study of several configurations (See Section VI). Then, conduct a larger and more realistic user study with human experts comparing the best performing configuration from the synthetic evaluation to the baseline (See Section IX). The advantage of the synthetic study is that it enables us to test each approach under exactly the same conditions.

VI. EVAL-1: SYNTHETIC STUDY

Synthetic study compares several configurations of our approach to the baseline VSM. In Section IV, we proposed both the Semantic Analysis (SA) and the TextRank (TR) in a combination with the baseline VSM. The objective of the synthetic study is to isolate the effects of both SA and TR: to test whether they improve over the baseline and whether they improve even further when both are combined.

B. Implementation Details

We have built a GUI-based tool named RETRO RANK as described in Figure 3. RETRO RANK takes as an input a user query (such as a set of keywords extracted from title and/or long description of bug report). As an output, RETRO RANK returns a ranked list of bug-fixing comments extracted from cross-cutting discussion threads from past resolved bug. RETRO RANK relies on MySQL database where it stores all OSS bug reports and OSS repositories. The UI platform is built upon well-known Python packages: i) tkinter that is a standard GUI package to build the framework and ii) sklearn used to build functionalities of RETRO RANK. In practice, RETRO RANK is designed to adapt to any commercial or OSS project that support modern development practices. The inclusion of 'pre-merge' code review [4, 39] is an example where relevant comments can be used as part of code review [20, 38]. Currently, RETRO RANK supports four OSS projects (i.e., GCC Compiler, Eclipse IDE, LibreOffice, and Apache Server). A screenshot in Figure 3 shows a list of recommended bug-fixing comments that can be used to address bug #61852. In this example, the RETRO RANK recommended several past fixed bugs (i.e., #26494, #40989, #53063, and #52304) and their associated bug-fixing comments shown in column 8;

A. VSM Model and its TR and SA Variants

We have used the VSM model implemented by Zhou et al. [71] as a baseline to compare our approach. However, the baseline alone does not take into account semantic relations or contextual overlap between keywords or the positive language used among comments. To complement this gap, we introduce the graph-based TextRank model (TR) as a variant of enhancing VSM model. Furthermore, because sentiment patters were noticeable in bug reports [13, 24, 30], we incorporate the Semantic Analysis (SA) to enrich and complement both the VSM model and the TextRank model. Hence, this study comparatively evaluates different variants of approaches. Those are: VSM (baseline), VSM+SA (our approach with TR disabled), VSM+TR (our approach with SA disabled) and VSM+SA+TR (our complete approach, in Section IV).

B. Research Questions

We ask two research questions towards our objective:

RQ1 What is the ranking performance of the baseline VSM, VSM+SA, VSM+TR, and VSM+SA+TR measured by the metrics we establish in Section VI-E?

RQ2 Are the effects of SA and TR orthogonal?

The rationale behind RQ1 is to measure the effects of SA and TR individually. Even if VSM+SA+TR outperforms VSM, it is possible that either SA or TR contributes more to the approach. Or, otherwise, preventing the approach from reaching its full potential. Likewise, we ask RQ2 because it is possible that SA and TR provide redundant effects.

C. Methodology

Our methodology to address research questions in the synthetic study is in four steps. First, we manually identify the goldset of bug reports and their corresponding bug-fixing comments; (Note that the goldset could represent bugs that are either in fixed status or duplicates). Second, we generate user queries for each bug report. Third, we use these ‘user-generated queries’ to run each configuration independently. Finally, we compute metrics to measure the performance of each configuration. WE discuss each of the steps as follows:
Step 1: To manually identify the goldset, first, we run a script to randomly select the 25 bugs that were in ‘FIXED’ status from four representative OSS projects as depicted in Table IX-E and detailed in Section IX-E (Note that, we deemed that this number of bugs was sufficient for performing a synthetic study that required extensive manual analysis of goldset data. Likewise, the analysis of the results from multiple combination of approaches (i.e., configurations); This sample was also comparable to other studies [1] [15]. Then, for each bug, we identified the bug-fixing comment(s) in the discussion thread that provided the actual fix of a bug—we flagged that comment or a set of comments to be as our goldset. In general, a comment that represents a clue or indicates a bug fix includes the revision link, attached patches, resolution steps or possible links to other duplicate bugs [5] [42] [71]. The revision alone may also include specific file names with their corresponding method signatures and explanations why changes were made. To make the final decision about the goldset, first author and a junior researcher resolved the conflicts of relevance using an established well-known majority-voting mechanism [2] [72]. Finally, the most highly voted comments were added in the goldset. In Table I, the goldset is denoted by ‘Bug ID’.

Step 2: Then, we recruited 12 external developers to create/generate the search queries for new bug reports, which do not represent the goldset examples. They are described in Table I, second column. Developers were asked to read these bug reports and generate as multiple search queries as possible (e.g., extract useful simple keywords from either the bug report title or the bug report long description that best described the root cause of that bug) as though they were searching for a bug-fixing comment. While we could have created these search queries ourselves, we felt that a safer and non-bias approach was to ask developers. Otherwise, we would inadvertently introduce a bias since we had already read the bug report conversations. Each developer read two bugs, with exception of one developer who read three bug reports due to odd number of bugs used for the evaluation. They were instructed to generate as many search queries as possible for that bug report. It should be noted that we only show those ‘user generated queries’ that led to the goldset. The bug reports were not repeated amongst developers. We could have tried multiple developers reading the same bug, but the chances for repeated queries to overlap would have been apparent.

Step 3: Next, we used the ‘user generated queries’ from Step 2 as input queries to run each configuration that we were testing (i.e., VSM, VSM+SA, VSM+TR, and VSM+SA+TR). It should be noted that the same ‘user-generated queries’ that were used as inputs to run the baseline VSM were also used to run other configurations under the same conditions. This was necessary to mitigate any bias towards retrieved results.

Step 4: Finally, once the results were retrieved from each configuration in Step 3, we then marked the ranking position where the goldset (i.e., bug-fixing comments from past resolved bugs) appeared for each configuration retrieved list. The positions are shown in Table I columns 3 to 6. Note that there could exist instances where more than one comment from the same bug discussion can be used as goldset as they may relate to the same topic in the conversation. We marked those occurrences with an asterisk sign. The criteria of measuring metrics and statistical tests are described in Section VII-E.

D. Participants

For step 2 in the previous section, we invited developers from three mid-sized software development companies by sending them an invitation survey. We categorized participants according to their level of experience. We selected 12 participants. Participants helped to create a dataset for the synthetic study. The participants did not offer subjective evaluations as they did for the user study. Also it should be emphasized that the participants from synthetic study and user study did not overlap. We view this as an advantage because it means a more diverse base of participants, which should reduce potential biases. All participants had more than five years programming experience. The preliminary online survey showed that about 91.7% of them were skillful in both C and C++. All of them reported high proficiency in Java. About 58.3% of them were familiar with the GCC Compiler. The majority of them, about 91.7% were familiar with Apache Server. While, 41.7% of them were familiar with LibreOffice, all of them knew Eclipse.

E. Metrics and Statistical Tests

The metrics to evaluate research questions RQ1 and RQ2 are established based on ranking and orthogonality.

RQ1 Ranking: To compute the ranking, the procedure was two fold: i) we counted the position where the goldset (i.e., identified bug-fixing comments) appeared in the retrieved list across all configurations, and ii) we compared their ranking position to evaluate which of the configurations performed better. It should be noted that the higher the position of the goldset comment, the better the performance of that particular configuration.

RQ2 Orthogonality: Similar to procedure described in RQ1, we evaluated the ranking differences of the results returned from our approach VSM+SA+TR against ranking differences of the results returned from variants VSM+SA and VSM+TR. This was necessary to investigate the effect of TR and SA impost on the performance between our approach and the baseline VSM.

Student’s $t$–Test [58] was used as a metric similar to other studies [1] [26] [64] to evaluate the statistical significance of differences in ranking and orthogonality of each configuration. We assumed on the null hypothesis that the mean values of the impact set sizes for all configuration were equal. Additionally, we leveraged standard ranked-based metrics similar to other studies [18] [50] [71] to enable a clear-cut comparisons among various approaches/configurations. Specifically, we measured mean average precision (MAP) and mean reciprocal rank (MRR) [22]. MAP considers average rank position of each relevant comment; MRR considers average position of the first relevant comment in the ranked list.
TABLE I: Performance between various configurations when ‘user generated queries’ formulated for new bugs are used as inputs for each configuration. Note that there exist instances of multiple levels of relevance of bug-fixing comments per bug as clues to investigate the new bugs. We denote these instances and their ranking positions in parenthesis and asterisks signs.

| OSS Project | Q | [Bug ID] [keyword1, keyword2, ...] | VSM | VSM+SA | VSM+TR | VSM+SA+TR | (Bug ID) | (Comment ID) |
|-------------|---|-----------------------------------|-----|-------|--------|-----------|--------|-------------|
| GCC GNU Compiler | Q1 | 60807 undo array destination | 12 | 2 | 2 | 1 | 9072 | C2 |
| | Q2 | 60807 wait compare warning | (12, 9)* | (6, 4)* | (2, 1)* | 43875 | (C9, C12)* |
| | Q3 | 61857 bracket-init-list | 6 | 6 | 6 | 47049 | C1 |
| | Q4 | 61850 lambda bugs | 10 | 4 | 4 | 56915 | (C9, C12)* |
| | Q5 | 61852 implicit declaration | (8, 12)* | (6, 4)* | (1, 5)* | 26949 | (C9, C12)* |
| | Q6 | 70009 implicit function declaration macro | (10, 12, 15)* | (4, 2, 1)* | (3, 2, 1)* | 71613 | (C9, C12)* |
| | Q7 | 61151 regression lambda | 13 | 6 | 6 | 47049 | C1 |

TABLE II: Statistical summary of the results from two evaluations. In Eval-1, ‘n’ denotes the 25 bug reports measured by the overall ranking performance. In Eval-2, ‘n’ denotes the number of comments measured by the overall relevance. µ denotes the average position of recommended comment(s) obtained by each configuration. For Eval-1, Student’s t-Test results are reported as p, t, t_crit, and Cohen’s d (effect size). For Eval-2, ANOVA results are reported in terms of F, F_crit, and p.

| Evaluation | Metric | Average Position of Recommended Comment(s) (µ) | t | F | F_crit | p |
|------------|--------|---------------------------------------------|---|---|--------|---|
| (Eval-1) Synthetic Evaluation | VSM+SA+TR vs VSM-SA | 1 | 2.0638 | 2.0638 | 0.1220 |
| | VSM-SA vs VSM+SA | 1 | 4.5E-75 | 20.818 | 2.9E-06 |

VII. EVAL-1: SYNTHETIC STUDY RESULTS

A. RQ1: Overall Ranking Performance

We found that VSM+SA+TR achieved the highest performance in terms of ranking position of the recommended bug-fixing comments, followed by VSM+TR, VSM+SA and the baseline VSM. In Table I, H₄, the average ranking position (µ) of the recommended bug-fixing for Config-5: VSM+SA+TR was (1.8) and for the baseline Config-5: VSM it was (9.1). In other words, on average, VSM+SA+TR placed the bug-fixing comments around the second or the first position, whereas the baseline VSM placed it in the ninth position. We also noticed that Config-4: VSM+SA and Config-3: VSM+TR placed the bug-fixing comments in a higher position than the baseline VSM for those configurations. In two cases, no approach located the bug-fixing comments. This most likely occurred due to no past comments found relevant to bug reports: #493305 or #33622. In only one case did the baseline VSM place the recommended bug-fixing comment in the first position. VSM+SA+TR placed the recommended bug-fixing comment in the first position for 16 instances of bug report queries. Results indicate that differences in average ranking position (µ) of the bug-fixing comments are statistically significant as reported in hypothesis H₄ (3.0E–11), Table I with p-value of 0.05 and a confidence level of 0.95 confirming the statistical significance. The Cohen’s d value [52] indicates a large effect size (d = 0.20165 > 0.8). That said, approaches by hypothesise H₅ are highly significant in terms of rankings. Similarly, we observe that both MRR and MAP scores are higher for VSM+SA+TR than VSM+TR or VSM+SA, and significantly higher than scores obtained for the baseline VSM. What that means is that our approach (VSM+SA+TR) significantly improves retrieval performance by 74% in terms of mean average precision (MAP) as denoted in Table I.

B. RQ2: Orthogonality

To answer RQ2, results indicated that both SA and TR provided orthogonal information. We observed that VSM+SA and VSM+TR outperformed the baseline VSM; In some instances, VSM+SA outperformed VSM+TR and vice-versa. If the approaches did not provide orthogonal information, we would expect that both approaches VSM+SA and VSM+TR most likely would not improve VSM+SA+TR. For example, we noticed that for queries Q₃, Q₄, Q₁₄, Q₁₅, Q₁₉, Q₂₀, Q₂₄, and Q₂₅, VSM+SA provided higher improvement than VSM+TR. Consider the example of Q₁₅, VSM+SA returns two comments (i.e., (C₉, C₁₂)*) from a same bug one position above (i.e., (2, 1)*) compared to VSM+TR which returns the same comments one position below for each comment (i.e., (3, 2)*). That said, the recommended bug-fixing comment
that were found during the query search ($C_9$, $C_{12}$) from bug 
#89404 were considered as candidate clues for addressing the 
LibreOffice bug #9932 and it was slightly better ranked 
by VSM+SA than VSM+TR. If we closely examine one of the 
recommended comment e.g., $C_9$ for query $Q_{15}$ in the 
boxed area below, it consists of words that are indicative 
of TR and SA representations. As noted in the text-box, 
VSM+SA has picked more positive words that are shown in 
bold: ‘resolved’, ‘fixed’, ‘fine’, ‘right’, ‘IMO’. In contrast, 
VSM+TR has picked only a few important lexical units that 
are underlined as: ‘saving and reopening’ and ‘hard recalc’. 

$C_9$: Doing ONE hard recalc both files work fine now... 
see Build ID c4e7d32d0d49397cad38d62472b0bc8aaff48dd6 
and linked patch saving and reopening after hard recalc keep 
the right values...There was several fixed for 4.4.0, fixed 
own. IMO this is resolved as worksform.”

These keyword provides positive affirmation for recommend-
ing clues to resolve a bug. Besides the positive keywords, the 
comment $C_9$ provides other information such as build link ID, 
patch link that are useful for bug localization.

VIII. THREATS TO VALIDITY (EVAL-1)

In the synthetic study, we identified three main threats to 
validity. The first threat is the use of 25 bug reports provided 
in the synthetic study as denoted in Table [I]. We could have 
completely omitted this step and only use the best performing 
configurations for the user study. However, our goal was 
to introduce a mixed-method to better assess the quality of 
different configurations used in the study. We followed a 
Goal-Metric-Question Paradigm [8] that is rarely practiced to 
answer research questions within the scope of bug localization. 
The second threat is the tasks themselves. We selected real 
bug reports from four well-known open-source repositories. 
The representation of bugs carry various complexities ranging 
from $P_{1}$ to $P_{4}$ priority bug reports. Often times these prioritized 
bug reports also require expert-knowledge. The query entries 
formulated by the participants could return content outside the 
scope of actual bug description. We attempted to mitigate this 
gap by allowing participants to formulate as many queries as 
possible when they were unsure about the requirements of the 
bug reports. The third threat is related to complexity of bug 
reports. Even though the majority of participants had more than 
five years of industry experience, there were certain bug 
reports that required domain knowledge. For example, a few 
participants struggled with specific GCC Compiler terminolo-
gies. We attempted to mitigate this threat by relying on our 
background questionnaire. We specifically asked participants 
about their expertise prior to completing any of their tasks.

IX. EVAL-2: USER STUDY

In the user study, we compare the best-performing approach 
VSM+SA+TR to the baseline VSM [71]. Our objective is to 
measure comment(s) relevance returned by each approach. 

### TABLE III: Cross-validation design of the user study.

| Experiment | Group | Search Tool | Task Set |
|------------|-------|-------------|----------|
| 1          | A     | VSM+SA+TR (RETRO RANK) | Bug01–Bug20 |
|            | B     | VSM (baseline) | Bug01–Bug20 |
| 2          | A     | VSM (baseline) | Bug21–Bug40 |
|            | B     | VSM+SA+TR (RETRO RANK) | Bug21–Bug40 |

A. Research Question

We ask the following research question to address relevance:

**RQ3:** To what extent do comments retrieved by the baseline 
VSM and RETRO RANK that combines VSM+SA+TR differ in relevance 
as perceived by the experts?

Research question RQ3 is intended to fill a gap that is left by 
the synthetic study. In the synthetic study, we found that when 
RETRO RANK combines VSM+SA+TR bug-fixing comment 
are substantially ranked in a better position than rankings 
obtained with VSM only. However, in the top-10 results from a 
search tool, all results may have some degree of relevance 
to the search query. So even though the baseline places the 
recommended answer at position 10, the first 9 results may 
provide useful information for developer to investigate the bug.

Intent for RQ3 is to measure the perceived relevance of the 
top-10 results of each tool—otherwise we could not be sure 
that one tool is outperforming the other.

B. Methodology

Our methodology for the user study was to ask participants 
to read bug reports, then use one of the approaches (VSM 
or VSM+SA+TR) to search for similar related bug-fixing 
comment(s) from past resolved bugs. To avoid biases, we 
used a cross-validation design to evaluate 40 real bug reports. 
Table III shows this design of two separate experiments. 
Prior to each experiment, we recruited 20 participants (not 
overlapping with the participants recruited to create a goldset 
in the synthetic study), and randomly divided these participants 
to two groups of 10 (denoted A and B). Participants were 
prevented from knowing whether they were evaluating our 
approach or the baseline approach to avoid introducing bias. 
Both tools were identical in both scenarios even in terms of 
UI look-and-feel features. During the study our approach was 
denoted as ‘Tool A’ and the baseline approach as ‘Tool B’. 
Once the tool naming situation was consolidated we asked in 
experiment 1, participants in group A to use VSM+SA+TR 
(a.k.a ‘Tool A’) to search for bug-fixing comments for bugs 
1-20. Participants in Group B used the baseline VSM (a.k.a 
‘Tool B’) for the same bug reports. Then, in experiment 2, 
group A used VSM and group B used VSM+SA+TR, but on a 
different set of 20 bug reports. The end effect was that 
different participants used different tools for different bug 
reports. During each experiment, a participant would read a 
bug report, formulate a query, and enter that query into 
the tool. Each participant was given one hour interval for 
each experiment to complete the bugs in each group. The 
participants then rated each of the top-10 results from each 
tool on a scale from [1–4] as we will describe in Section IX-F.
TABLE IV: Characteristics of the open-source projects

| OSS Project           | Project Description                                      | Interval          | Product Name | Component Type | #Bug Reports | # Resolved Bugs | # Unresolved Bugs |
|-----------------------|----------------------------------------------------------|-------------------|--------------|----------------|--------------|----------------|------------------|
| GCC GNU Compiler      | Compiler system produced by the GNU project             | Aug 1999 – Aug 2018 | GCC          | C and C++      | 75901        | 66608          | 9293             |
| Eclipse IDE           | Development platform for Java language                  | Mar 2002 – May 2017 | Platform IDE | IDE            | 8028         | 4321           | 3707             |
| LibreOffice           | Office used for document creation                       | Oct 2010 – May 2018 | LibreOffice  | C++            | 47336        | 1757           | 27540            |
| Apache Server         | Web server platform on the HTTPd server                 | Jan 2002 – Apr 2019 | HTTPD-2      | CORE           | 2511         | 19796          | 754              |

Total #Bug Reports: 133776, #Resolved Bugs: 92482, #Unresolved Bugs: 41294

Figure 3 depicts the rating scale each participant would select depending on which comment would be more relevant to addressing the new bug report. The cross-validation design follows the norms established in software engineering. We asked participants to evaluate the top–10 results due to the accepted practice that researchers follow [19, 50, 69].

C. Participants

We recruited 20 participants to conduct the user study. Half of the participants in the study had industry experience. The rest of the participants were recruited from academia. Based on a preliminary online survey, 81% of the participants had more than 3 years of programming experience. From all participants, 90.5% of them were skillful in C, C++, and Java. About 40% of their current job was in software development. In addition, 57.1% of the them asserted their experience with issue-tracking systems. Participants distributed their experience among four projects as follows: i) 52.4% of them were familiar with GCC Compiler, ii) 85.7% were familiar with Eclipse IDE, iii) 33.3% were familiar with Apache Server, and iv) Lastly, 42.9% of them were familiar with LibreOffice.

D. User Study Tasks

We designed the experiments by providing 40 bug reports as the main tasks for the user study. To avoid introducing any bias in the study, the bug reports were real-world examples randomly selected from four distinct open source projects as reported in Table IV and Section IX-E. All 40 bugs were identical to the ones found via online issue-tracking systems. These bugs varied in terms of priority [P1–P3]. In bug report prioritization, P1 and P2 are categorized as high priority bugs whereas P3 and P4 are categorized as low priority bugs that do not require immediate fix.

E. Subject Open-Source Projects

The bug repositories of open-source projects are considered the main source to evaluate RETRO-RANK and the baseline. Table IV reports four different open source software systems. Those are: GCC GNU Compiler, Eclipse IDE, Apache Server, and LibreOffice. All four projects include a full repository of bug reports filtered by product and component type. Our decision to select these four projects is based on four premises. First, all projects fall within three categories of small, medium, and large projects. Hence, they are generalizable across domains. Second, all projects use bug tracking systems and version control to easily trace code changes. Third, all projects consist of bug reports containing lengthy discussions with hundreds of comments. Finally, they are extensively used in software engineering research community.

F. Metrics and Statistical Tests

The metric to evaluate the research question RQ3 is as follows: RQ3. Relevance: Comments were evaluated by the participants on a four-point Likert scale from [1-4] where four is ‘highly relevant’, three is ‘relevant’, two is ‘somewhat relevant’ and one is ‘not relevant’. To compute the metric for RQ3, we used the comment relevance score metric.

Our criteria for selecting comment relevance is as follows:

1) Not Relevant (1)—the information the comment contains does not include any relevant information to query input.
2) Somewhat Relevant (2)—the information the comment contains, mostly lacks details to references to other artifacts or links, the explanations may be useful for partial understanding or limited bug investigation clues.
3) Relevant (3)—the information the comment contains reveals details such as: artifact links that trace the bug, and to some extent, commit patches with partial explanations.
4) Highly Relevant (4)—the information the comment contains reveals extremely useful details such as: code snippets and patches with diffs, links to related/duplicate reports, resolution steps and change explanations.

X. EVAL-2: USER STUDY RESULTS

This section describes results from our user study, as well as our conclusions and answer to RQ3.

RQ3: Overall Relevance. Results show that RETRO-RANK when using VSM+SA+TR achieved higher relevance than when the baseline VSM was used alone [71]. The evidence is reported in Table II RETRO-RANK (VSM+SA+TR) based on statistical significance reported by hypothesis H2. Note that the obtained statistical test results are based on metrics defined in Section IX-F. A concrete example is also described in Table V. Due to space limitations we only show results obtained by two participants. The key data in Table V is the ‘Relevance Score’ column which shows the ranking scores from [1–4] as defined in Section IX-F. The relevance scores that participant P9 in (group A) selected for RETRO-RANK (VSM+SA+TR) were significantly higher than the relevance scores that participant P9 in (group B) selected for VSM [71]. We also observe a universal pattern across all bugs with the relevance scores from both approaches. For example, for the baseline VSM, the relevance scores tend to decrease within the top-10 query but for RETRO-RANK (VSM+SA+TR) tend to increase within the top-10 query results. Another evidence found in this data is that the recommended comment that is placed in the first position is marked as ‘highly relevant’.
TABLE V: Retrieval and relevance scores of comments returned by both our approach VSM+SA+TR and the baseline VSM. These results are reported from the user study completed by two participants chosen from two groups to address Bug ID:61852.

| Approach       | Project | Short Description                          | User Query Input                                                                 | Query Output Results                                                                 | Bug #, Comment | Relevance Score [1-4] |
|----------------|---------|--------------------------------------------|---------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|----------------|------------------------|
| VSM+SA+TR      | GCC     | Incorrect column number for implicit-function-declaration | Wimplicit function declaration                                                  | incorrect column number for wimplicit-function-declaration                           | 9738, C9      | 4                      |
|                |         |                                            |                                                                                 | GCC                                                                                  |                |                        |
| VSM            | GCC     | Incorrect column number for implicit-function-declaration | Wimplicit function declaration                                                  | internal compiler error in rest_of_compilation at toplevel.c:3491                   | 9738, C9      | 1                      |
|                |         |                                            |                                                                                 | the compiler crashes                                                                 | 5830, C10      | 1                      |

However, we do not perceive a similar outcome from the results returned by the baseline VSM. Additionally, the query results from RETRO RANK (VSM+SA+TR) show a stronger relationship with the content from the query input and bug description. To illustrate, in all cases where relevance score is either 3 or 4, the results consist of vocabulary that refer to compiler signaling warnings. Hence, we pose that the differences in relevance of comments are statistically significant between RETRO RANK (VSM+SA+TR) and the baseline VSM. Such evidence is provided in Figure 7, hypothesis H6, (4.5E−75) with p-value of 0.05 and a confidence level of 0.95. The average relevance scores for RETRO RANK (VSM+SA+TR) (μ = 2.7) are higher compared to the average relevance scores for the baseline VSM (μ = 1.4).

XI. Threats to Validity (Eval-2)

There are several threats that impact the validity of our work. We focus on internal and external threats.

Internal Threats: The internal threat of a study is the extent to which a treatment effects change in the dependent variable. In this experiment it could be due to the content of the comment. If a comment lacks informativeness [9, 70], the outcome of RETRO RANK could be affected. Other factors may relate to individual differences among subject groups. A few participants more than the others may be influenced by factors of stress, tiredness or programming experience. We attempt to minimize these threats by alternating the order of tasks at each participant’s turn using a cross-validation design. We further instruct subjects not to discuss the experiments with each other.

External Threats: The external threat of a study is the extent to which we can generalize the results of our user study. Our evaluation focused solely on OSS bug reports. A threat that might limit the generalizability of our results is the use of only 40 real-world bug reports. We attempt to minimize this representative sample by selecting bug reports from four well-known open source domains. To mitigate further threats, we also engaged subjects from academia and industry to help generalizing our findings to both contexts.

XII. Related Work

Current studies have developed various methods that focus on automatically ranking source files for given bug reports. Kim et al. [25] applied a two-phase machine learning approach to predict files that could fix a bug report. DebugAdvisor [6] used searches over structured and unstructured text to recommend a set of similar past tasks and the files that were changed. Saha et al. [54] presented BLUiR and used the structure of bug reports and source code files to build a retrieval structure for reporting the bugs. Nguyen et al. [41] applied LDA model to predict buggy files for given bug reports. Similarly, Lukins et al. [29] applied LDA to predict and filter the top 10 defective files across 22% of overall bug reports. Ye et al. [68] used developer domain knowledge (e.g., API documents) to identify source files associated to a bug. The same authors proposed a ranking model to recommend top 10 source files [69]. Zhou et al. [71] used VSM model to recommend textual similarities among similar past bugs and their associated files to address a new bug. Our approach instead recommends similar past bugs by extracting relevant bug-fixing comments from lengthy discussion threads of past bugs to address a new bug. A line of research by Runeson et al. [53], Wang et al. [67], Bettenburg et al. [10], Sun et al. [59], Hindle and Onuczko [22] focused on detecting duplicate reports by querying bug reports.

XIII. Conclusion and Future Work

In this paper, we introduced a novel approach for recommending related bug-fixing comments from cross-cutting discussion threads of past solved bugs in support of bug repair. It is a common practice that when the developers investigate a new bug they often need to know how similar bugs were solved in the past. Our approach was based on both a combination of Sentiment Analysis and the TextRank semantic-based techniques. By incorporating both combinations into a text similarity strategy (VSM), we identified bug-fixing comments that were more positive and semantically connected. Additionally, two evaluations demonstrated that both types of information improved search, relevance. It also provided insightful summary of bug repair instructions. For future work, our goal is to perform a systematic qualitative study where the rationale behind the achieved results will be shown with real scenarios of bug-fixing examples explained in greater detail.
[42] Y. Noyori, H. Washizaki, Y. Fukazawa, K. Ooshima, H. Kanuka, S. Nogiri, and R. Tsujiya. What are good discussions within bug report comments for shortening bug fixing time? In 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), pages 280–287. IEEE, 2019.

[43] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[44] J. H. Paik. A novel tf-idf weighting scheme for effective ranking. In Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, pages 343–352. ACM, 2013.

[45] B. Pang and L. Lee. Opinion mining and sentiment analysis. Foundations and Trends in IR, 2(1–2):pp. 1–135, 2008.

[46] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In Proceedings of the ACL-02 conference on Empirical methods in natural language processing, pages 79–86. Association for Computational Linguistics, 2002.

[47] M. F. Porter. An algorithm for suffix stripping. Program, 14(3):pp. 130–137, 1980.

[48] D. R. Radev, E. Hovy, and K. McKeown. Introduction to the special issue on summarization. Computational linguistics, 28(4):399–408, 2002.

[49] J. Ramos et al. Using tf-idf to determine word relevance in document queries. In Proceedings of the first instructional conference on machine learning, volume 242, pages 133–142, 2005.

[50] M. Rath, D. Lo, and P. Mäder. Analyzing requirements and traceability G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. Commun. ACM, 18(11):613–620, 1975.

[51] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate bug reports using natural language and execution information. In Proceedings of the 15th International Conference on Mining Software Repositories, pages 442–453, 2018.

[52] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pages 499–510, 2007.

[53] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 345–355. IEEE, 2013.

[54] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. Inf. Process. Manage., 24(5), 1988.

[55] G. Salton, A. Singhal, M. Mitra, and C. Buckley. Automatic text structuring and summarization. Advances in automatic text summarization, pages 342–355, 1999.

[56] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. Commun. ACM, 18(11):613–620, 1975.

[57] 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 Proceedings of the 34th International Conference on Software Engineering (ICSE), pages 14–24. IEEE, 2012.

[58] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, pages 623–632. ACM, 2007.

[59] C. Sun, D. Lo, S. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011, pages 253–262, 2011.

[60] F. Thung, P. S. Kochhar, and D. Lo. Dupfinder: integrated tool support for duplicate bug report detection. In Proceedings 29th International Conf. on Automated Software Engineering, pages 871–874, 2014.

[61] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. Learning to rank relevant files for bug localization using machine learning techniques. In Proceedings of the 22nd ACM Intnl Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pages 689–699, 2014.

[62] X. Ye, R. C. Bunescu, and C. Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. IEEE Trans. Soft. Eng., 42(4):379–402, 2016.

[63] X. Ye, R. C. Bunescu, and C. Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. IEEE Trans. Soft. Eng., 42(4):379–402, 2016.

[64] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia. Bug report enrichment with application of automated fixer recommendation. In Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017, pages 230–240, 2017.

[65] 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 Proceedings of the 34th International Conference on Software Engineering (ICSE), pages 14–24. IEEE, 2012.