BugSum: Deep Context Understanding for Bug Report Summarization

Haoran Liu, Yue Yu*, Shanshan Li, Yong Guo, Deze Wang, Xiaoguang Mao
{liuhaoran14,yuyue,shanshanli,yguo,wangdeze14,xgmao}@nudt.edu.cn
College of Computer Science and Technology, National University of Defense technology
Changsha, Hunan, China

ABSTRACT

During collaborative software development, bug reports are dynamically maintained and evolved as a part of a software project. For a historical bug report with complicated discussions, an accurate and concise summary can enable stakeholders to reduce the time effort perusing the entire content. Existing studies on bug report summarization, based on whether supervised or unsupervised techniques, are limited due to their lack of consideration of the redundant information and disapproved standpoints among developers' comments. Accordingly, in this paper, we propose a novel unsupervised approach based on deep learning network, called ***BugSum***. Our approach integrates an auto-encoder network for feature extraction with a novel metric (***believability***) to measure the degree to which a sentence is approved or disapproved within discussions. In addition, a dynamic selection strategy is employed to optimize the comprehensiveness of the auto-generated summary represented by limited words. Extensive experiments show that our approach outperforms 8 comparative approaches over two public datasets. In particular, the probability of adding controversial sentences that are clearly disapproved by other developers during the discussion, into the summary is reduced by up to 69.6%.

CCS CONCEPTS

• **Software and its engineering** → Software maintenance tools; • **Collaborative and social computing** → Collaborative and social computing systems and tools.

KEYWORDS

Bug Report Summarization, Deep Learning, Software Maintenance, Mining Software Repositories

ACM Reference Format:
Haoran Liu, Yue Yu*, Shanshan Li, Yong Guo, Deze Wang, Xiaoguang Mao. 2020. BugSum: Deep Context Understanding for Bug Report Summarization. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389272

*Corresponding author.

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389272

1 INTRODUCTION

There is a golden rule that originated in the open source movement, *i.e., given enough eyeballs, all bugs are shallow* [44]. Inspired by this rule, both open source and commercial projects [19] tend to manage their development tasks using bug repositories (*e.g., Bugzilla* [1]) or issue tracking system (*e.g., Issue Tracker in GitHub* [2]) online. A large number of bug reports and discussions have been accumulated, which are valuable for software long-term maintenance [2] and developer onboarding [7].

In modern software development, a bug report is usually organized in the form of as an open post with a discussion section akin to those on social web sites [20]. When contributors find bugs in the software, they submit bug reports asked for a basic description about the exposed issue using natural language [3, 11, 53]. Then, other stakeholders, including project managers, maintainers or external contributors, discuss and likely put forward different standpoints to the issue in the form of comments. This process evolves dynamically along with the development of the project, *i.e.,* the original bug report and all discussions can be read by any stakeholder, who may evaluate and directly reply to those standpoints with their own opinions based on the development status, also in the form of comments. Thus, the scale of a bug report increases rapidly through continuously iterative discussion [42]. According to our statistics, which are based on the dataset described in **Section 2**, 25.9% of bug reports contain more than 15 comments, while each comment contains 39 words on average. Therefore, an accurate and concise summary can effectively reduce the time consumed in wading through all posted comments [42]. While modern bug repositories (*e.g., Debian* [3]) encourage contributors to manually write a summary for each bug report, only 2.80% of the bug reports in our dataset were found to have been equipped with artifact summaries using 29 words on average, which is insufficient to furnish stakeholders with the required information.

Bug summarization has been proven to be a promising method [43] of auto-generating summaries by selecting salient sentences based on supervised [17, 43] or unsupervised [22, 30, 52] machine learning techniques. The performance of traditional supervised approaches relies heavily on the quality of the training corpus [27], which requires massive manual effort and annotators with certain expertise. Additionally, existing unsupervised approaches rely excessively on word frequency, which tends to introduce extra bias for two main reasons: 1) bug reports are **conversation-based** texts with frequent **evaluation behaviors** [27], *i.e.*, the discussions are cross-validated among different

[1] https://www.bugzilla.org/
[2] https://help.github.com/en/github/managing-your-work-on-github/about-issues
[3] https://www.debian.org/Bugs/server-control#summary

stakeholders according to their own experiences, and some comments will be disapproved by other participators; 2) a group of comments supporting the same standpoint have similar semantic features, so word frequency-based approaches would be more likely to introduce redundant sentences (*i.e.*, sentences that are different but represent a similar topic) into the auto-generated summary, which will decrease its comprehensiveness due to the word length limitation.

Accordingly, in this paper, we propose a novel unsupervised approach of bug report summarization, called ***BugSum***. Firstly, we use a trained auto-encoder network to extract semantic features by converting the sentences to vectors. Meanwhile, we design a novel metric (*i.e.*, *believability score*) to measure the degree that a sentence has been approved against disapproved among the interactive discussions. Then, for each sentence, we merge the believability score into the vector to reduce the possibility of controversial sentences being selected into the summary. Finally, we optimize the summary auto-generation process using a dynamic selection strategy that considers both informativeness and comprehensiveness.

We compared ***BugSum*** with eight comparative summarization approaches over two public datasets. Experimental results demonstrate that our work outperforms comparative approaches in terms of the metrics that have been widely used in previous studies.

The key contributions of this paper include:

- We proposed a novel unsupervised bug summarization approach by integrating the auto-encoder network, sentence believability assessment and dynamic selection.
- The probability of adding controversial sentences, which are clearly disapproved by other developers during discussion, into the summary is reduced by up to 69.6% according to our careful manual evaluation.
- We design extensive evaluations on two public datasets to demonstrate that our approach achieves the state-of-the-art performance. It outperforms 8 comparative approaches by up to 0.166 and 0.117 in terms of *F-score* and *Rouge-1* metrics respectively.

2 MOTIVATION

A high-quality bug report can comprehensively expose the software bug *per se*, while also recording all participants' discussions according to the following process. A contributor first describes the **issue** and submits it as the **description** part of a bug report using natural language. Subsequently, other stakeholders, including project managers, maintainers or external contributors, discuss the proposed issue in the form of comments. The content of these discussions include aspects such as steps to reproduction, issue location, and possible solutions. During the discussion, different standpoints are likely to be proposed, *e.g.*, *comment#8* and *comment#11* in Fig. 1. These standpoints can be read by other stakeholders, who may evaluate and directly reply to these standpoints to express their own opinions (*i.e.*, **evaluation behaviors**), also in the form of comments. Such interaction structure has been referred to as conversation-based text in previous work [17, 43]. Here, we refer to a group of comments interconnected by evaluation behaviors as a **conversation**. It has been proven that sentences in conversations contain a large amount of important information such as observed bug behaviours, steps to bug reproduction, and possible solutions [1, 27]. This information is not contained in the description, and need to be included into the summary.

In order to develop a thorough understanding of the conversations in bug reports, we analyzed 31,155 bug reports from 7 popular OSS projects (6,954 Hadoop, 1,177 ZooKeeper, 5,705 Derby, 1,826 Yarn, 8,876 Flink, 3,710 HDFS and 2,907 Hive). To ensure the complete life-circle information of bug reports is included, all those bug reports are selected with the status of "Closed". We summarize our findings in the following two aspects, *i.e.*, believability and redundancy of the sentences among conversations.

2.1 Salience and believability

Typical summarization approaches focus on identifying salient sentences from bug reports and conversations, based on word frequency [40], predefined structure [27] and so on. However, in our dataset, we found that 71.5% of the bug reports contain evaluation behaviors that express attitudes of approval or disapproval within their conversations, covering over 36.4% of sentences on average. It is a significant challenge to balance the salience and believability of these highly discussed sentences for traditional approaches.

For example, as shown in Fig. 1, the report describes a system bug reading *"Resource temporarily unavailable"* in the last sentence. Subsequently, *Comment#8* proposes a solution, which is approved by *Comment#10* but disapproved by *Comment#11*. Moreover, *Comment#11* explains the reason for the disapproval and proposes another solution, which is approved by *Comment#13*. In brief, the standpoint in *Comment#8* is controversial in the conversation. The more a standpoint (comment) is disapproved by others, the lower believability it has, and vice versa. In addition, we define such comments that are disapproved by at least one other comment as **controversial comments**, and the sentences within these comments as **controversial sentences**. Words and sentences related to the controversial comments have a relatively high appearance in the bug report, as stakeholders may discuss the standpoint being proposed by the controversial comment for several rounds. Thus, controversial sentences are highly preferred by previous word frequency based approaches, *e.g.*, Centroid [40], which would introduce a significant bias into the summary.

Thus, when we attempt to generate a high-quality summary, the salience of a candidate sentence is determined by the information it contains (*i.e.*, *informativeness*) and the extent to which the standpoint is believable (*i.e.*, *believability*).

2.2 Redundancy in Sentences

Bug reports are conversation-based text. Sentences in the same conversation that discuss relevant standpoints tend to contain redundant semantic features, such as specific domain keywords related to the discussed standpoint.

Starting from our dataset, we vectorize sentences using the Bag-of-Words strategy, and calculate the cosine similarity to measure the semantic similarity between sentences. The result shows that, on average, sentences in the same conversation are 17.4% more similar than sentences spread in different conversation. This means

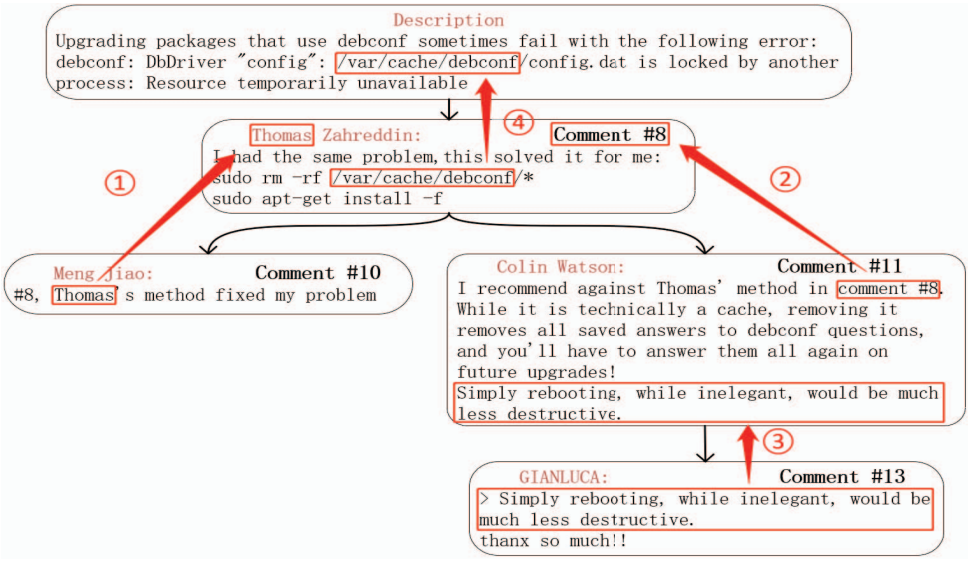


Figure 1: Evaluation behavior among sentences.
(Bug #349467 of deconf from launchpad, https://bugs.launchpad.net/debconf/+bug/349469/)

that sentences within the same conversation have relatively high semantic redundancy. If one sentence in a conversation has been assigned a relatively high salient weight (*e.g.*, measured by word frequency [22, 40]), the other sentences in the same conversation may also have similar high scores due to the word similarity. Since sentences with higher salience are more likely to be selected by previous approaches, the bug report summary generated by the sentences in the same conversation would hardly conclude the entire bug report because of the semantic redundancy [13].

3 BUGSUM DESIGN

We design a novel unsupervised approach **BugSum**. BugSum uses an auto-encoder network to obtain the domain textual features in sentences. An assessment is deployed to evaluate the believability of sentences. Finally, BugSum generates summaries under certain word amount limitation through a dynamic selection of informative and believable sentences, while considering the comprehensiveness of the selected sentences.

As shown in Fig. 2, BugSum consists of four steps: *Bug Report Pre-processing*, *Sentence Feature Extraction*, *Sentence Believability Assessment*, and *Summary Generation*. Bug report pre-processing removes noises from bug reports and divides bug reports into sentences. Sentence feature extraction further compresses sentences into sentence vectors, while sentence believability assessment assigns believability scores to sentences. BugSum constructs a full-text vector through weighted combining all sentence vectors in the bug report. At last, the summary generation step dynamically selects salient sentences from the bug report to form a summary.

3.1 Bug Report Pre-processing

Sentences in a bug report as real-world data contain a considerable amount of noises [49], meaning that a pre-processing step for noise removal is required.

During pre-processing, BugSum divides a bug report into sentences based on punctuation marks such as '.', '!', '?', and ';', apart from when the punctuation is used as a part of a string. Function names such as "book.find", which include the words "book" and "find", are treated as new words. BugSum tokenizes these sentences using the software-specific regular expression [27] to preserve the majority of the function and variable names while correctly identifying words. BugSum further stems these tokens using the poster stemmer [39] and removes the stop words [10]. In this step, each bug report is divided into sentences for further processing.

3.2 Sentence Feature Extraction

Sentence feature extraction is deployed to extract domain features from sentences. In BugSum, we use a trained auto-encoder network to generate sentence vectors with domain features. The structure of auto-encoder network is illustrated in Fig. 2(B). The auto-encoder consists of an encoder and decoder. The basic principle of the auto-encoder is that each sentence is encoded into a vector, after which it is decoded by the decoder to reconstruct the original input sentence. The more consistent the input is with the output, the more precisely the vector can express the textual features of the sentence.

The encoder of the network processes one sentence at a time. The words in the sentences are first embedded into word vectors, after which these word vectors are recurrently processed by the recurrent units of the encoder. BugSum uses the last state of the encoder as the feature vector of the input sentence.

To preserve both the forward and backward contextual features of each sentence, we employed bidirectional GRU (Bi-GRU) [51] as the recurrent unit. Bi-GRU consists of a forward GRU and a backward GRU, which takes the word embedded sequence forward and backward, respectively. The encoder concatenates the last hidden states of both the forward and backward GRU to form a sentence vector. We denote the sentence vector of sentence i as S_i.

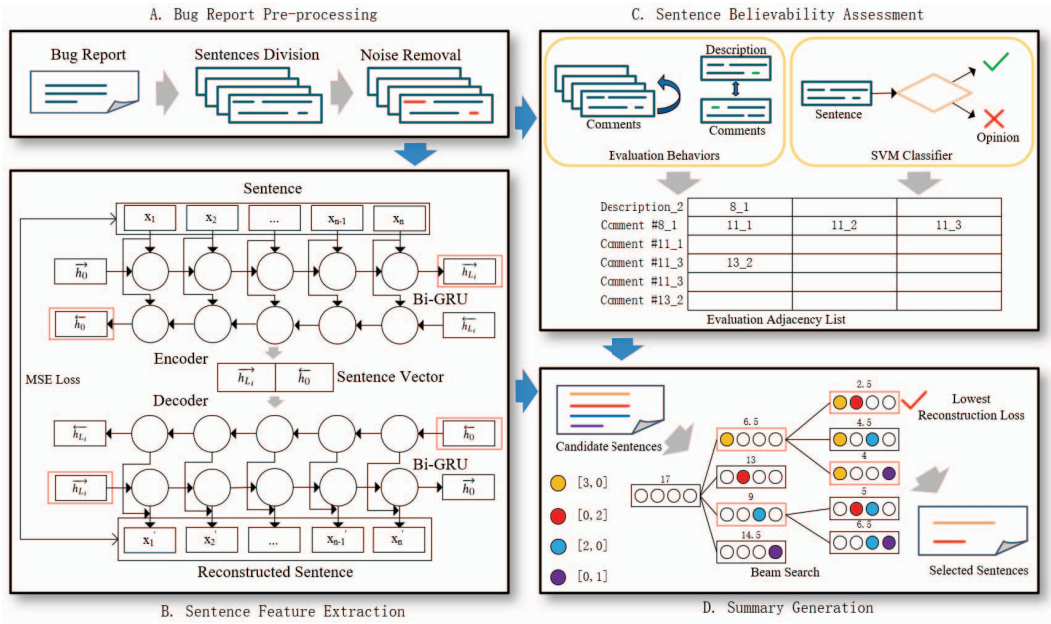


Figure 2: The framework of BugSum.

We pre-process the dataset we discussed in Section 2 to create the training set. During the training process of the auto-encoder network, we minimize the MSE loss [9] between the input sentence and its corresponding decoded sentence. The parameters are optimized using the widely used SGD optimizer [4], and the learning rate is set to 0.01 [21]. We continue this optimizing process until the MSE loss is minimized and remains stable.

3.3 Sentence Believability Assessment

As was discussed in Section 2.1, the importance of sentences in the bug report is determined by their informativeness and believability. BugSum captures the evaluation behaviors in bug reports, and computes sentence believability based on these evaluation behaviors. BugSum further uses the obtained believability scores as weights and accumulates the weighted mean of the sentence vectors to get the full-text vector. The full-text vector combines the believability of the sentences and domain textual features in the entire bug report.

3.3.1 Evaluation Behaviors. There are different expressions of evaluation behaviors in bug reports. Sentences in comments are usually evaluated in the form of replies, while sentences in the description are usually evaluated in the form of domain word sharing. BugSum captures these evaluation behaviors based on their expressions.

As shown in Fig. 1, arrows 1, 2, and 3 demonstrate the evaluation behaviors occurring within comments, which are usually explicit replies. These replies often clearly indicate the person's name or the comment number of the evaluated comment, or quote the sentences being evaluated. Same to the comments, the description is evaluated by stakeholders. Sentences in the description may be confirmed, discussed, or extended during the evaluation process. As the description usually not be replied directly, evaluation behaviors between the description and comments usually take the form of domain word sharing. As indicated by the arrow 4 in Fig. 1, the description and comments sharing domain words will be considered as having evaluation behaviors.

When considering the evaluation behaviors among comments, BugSum only focuses on the explicit replies. Ambiguous replies, which do not have an explicit target, are ignored because we cannot ensure the correctness of the deduction. We store the set of sentences that evaluate sentence i in the evaluation adjacency list, and denote this set as $EAdj_i$. As for the evaluation behaviors between the description and comments, we use *TF-IDF* [41] as the word assessment method in BugSum, and select words with the TF-IDF scores higher than a threshold as domain words. The threshold is denoted as θ and will be tuned in Section 4.4.2. Sentences in the description and comments that share domain words will be considered as evaluation behaviors. BugSum takes such evaluation behaviors and stores them in the evaluation adjacency list. Finally, we obtain an evaluation adjacency list that contains evaluation behaviors in the bug report.

3.3.2 Believability Score Assignment. Sentences in the bug report are supported or disapproved by other sentences during the discussion, such evaluation behaviors cause sentences to have differing believability. Sentences supported by other sentences are more believable, while controversial sentences are likely to be incorrect. BugSum uses evaluation behaviors to assess how believable a sentence is to be selected into the summary.

We denote the believability score of sentence i as $Bscore_i$. The believability scores of sentences are further modified based on their related evaluation behaviors. Since we use multiplication when calculating the believability of a sentence, the believability score of each sentence is initialized to 1 so that it remains neutral during the calculation.

$$Bscore_i = \begin{cases} 1 + \sum_{j \in EAdj_i} (Bscore_j * OPscore_j), & |EAdj_i| > 0 \\ 1, & |EAdj_i| = 0 \end{cases} \quad (1)$$

The number of sentences that evaluate sentence i is denoted as $|EAdj_i|$. When $|EAdj_i| = 0$, this indicates that sentence i is not evaluated by any other sentences, so its believability score remains 1. When $|EAdj_i| > 0$, the believability score of the evaluated sentence i is modified by its evaluator sentences in $EAdj_i$. The weight of each evaluator sentence is decided by its believability score and its opinion on the evaluated sentence. BugSum uses opinion scores to assess the opinions of evaluator sentences towards the evaluated sentences. We denote the opinion score of sentence i as $OPscore_i$. Each opinion score is assigned via a pre-trained Support Vector Machine (SVM) classifier [45]. We train the SVM classifier over a dataset containing 3000 sentences, these sentences are collected from the dataset mentioned in Section 2 and manually labeled. Half of these sentences exhibit a negative opinions on the evaluated sentences. The SVM classifier takes a sentence as an input and predicts the possibility that it expresses a *negative opinion*, taking a value between 0 and 1. To facilitate the calculation, we subtract the possibility from 0.5 and then multiply by 2, and use it as the sentence opinion score, whose value is between -1 and 1. When the sentence i is evaluated by sentence j, and the value of $OPscore_j$ is less than 0, it means that the sentence i is possibly disapproved by sentence j. Therefore, sentence i is a controversial sentence, and its believability score $Bscore_i$ will decrease according to Formula 1. Otherwise, $Bscore_i$ will increase.

If sentences are disapproved by most of its evaluator sentences, their believability scores may be lower than 0. Under these circumstances, we set their believability scores to 0. The reason is that, if sentence j that has a low believability ($Bscore_j < 0$) also has a negative opinion ($OPscore_j < 0$) regarding sentence i, the value of formula $Bscore_j * OPscore_j$ will be greater than 0, meaning that $Bscore_i$ will increase according to Formula. 1. It is incorrect because a sentence disapproved by a incorrect sentence is not necessarily correct.

$$Bscore_i = max(Bscore_i, 0) \quad (2)$$

For each bug report, BugSum takes the sentences believability scores as weights, and sums the weighted average of the sentence vectors to obtain the full-text vector *i.e.*, DF.

$$DF = \sum_{i=1}^{n} Bscore_i * S_i \quad (3)$$

where n is the amount of sentences in the bug report, and S_i is the sentence vector of sentence i. Therefore, DF represents the domain features of the bug report, as it combines the domain textual features with the believability scores of all sentences.

3.4 Summary Generation

Since it is simpler and more effective to apply the extractive technique [27], we apply this technique for Bugsum, which selects salient sentences from the bug report to form a summary. The selected sentences should be able to preserve the domain features of the entire bug report as comprehensively as possible. Therefore, during the selection, the **informativeness** of sentences should be determined by their ability to improve the **comprehensiveness** of the selected sentences.

Algorithm 1 Beam Search.

Input: Sentences set S, full-text vector DF, beam size b, word amount limitation θ
Output: A sentence set *Chosen* with the highest δ

$L_{new} \leftarrow \phi, L_{old} \leftarrow S, L_{Chosen} \leftarrow \phi$
while L_{old} is not empty **do**
  for Each sentence set l_i in L_{old} **do**
    for Each sentence s_j in S **do**
      if $s_j \notin l_i$ **then**
        $l_{new} \leftarrow l_i \cup s_i$
        if word amount of l_{new} $len(l_{new}) < \theta$ **then**
          $\delta \leftarrow$ Similarity between $\widetilde{DF}$ and DF
          if l_{new} can't be further extended **then**
            Append l_{new} to L_{Chosen}
            Update L_{Chosen} to reserve top-b sentences set with lowest δ
          else
            Append l_{new} to L_{new}
            Update L_{new} to reserve top-b summary set with lowest δ
          end if
        end if
      end if
    end for
  end for
  $L_{old} \leftarrow L_{new}$
  $L_{new} \leftarrow \phi$
end while
Select *Chosen* from L_{Chosen} with the lowest δ
return *Chosen*;

If k sentences are selected from the bug report, we denote the set consisting of selected sentences as *Chosen* and the dimension of the feature vector as d. BugSum uses the selected sentences to reconstruct the full-text vector $\widetilde{DF}$, and subsequently uses the Mean Squared Error (MSE) between $\widetilde{DF}$ and DF to represent the reconstruction loss, which is indicated as δ.

$$\widetilde{DF} = \sum_{i \in Chosen} Bscore_i * S_i \quad (4)$$

$$MSE = \frac{\sum_{i=1}^{d} (y_i - \widetilde{y_i})^2}{d} \quad (5)$$

$$\delta = |MSE(DF, \widetilde{DF})| \quad (6)$$

A lower value of δ indicates that, the selected sentence set contains more domain features of the entire bug report, *i.e.*, the selected sentence set is more comprehensive. BugSum aims to find the optimal set of sentences that minimizes δ under the word amount limitation θ. This can be seen as a generalization of the knapsack problem, which has been proven to be NP-hard [24]. A bug report with n sentences can generate 2^n different summaries,

making it extremely inefficient to evaluate all possible combinations. Greedy algorithms (*e.g.*, beam search algorithm) are effective approximate approaches to solve NP-hard problem. The beam search algorithm greedily traverses the entire candidate set recurrently, and looks for the top-b choices that can maximize the improvement in each iteration. It can significantly reduce the time effort compared to evaluate all possible combinations.

The reconstruction loss δ can leverage the comprehensiveness of selected sentences during the selection process. For example, in Fig. 2(D), the bug report contains 4 sentences, and the dimension of the feature vectors is 2. We denote their sentence vectors as $S_1 = [3, 0]$, $S_2 = [0, 2]$, $S_3 = [2, 0]$ and $S_4 = [0, 1]$. All of them have the same believability score, which is 1. The full-text vector can be calculated according to Formula 3, from which we can get $DF = [5, 3]$. The feature represented by the first dimension of the vector appears more frequently in the bug report, which means that it is likely to be more important. When the candidate set is (S_1), the reconstructed full-text vector can be calculated as $\widetilde{DF_1} = [3, 0]$ based on Formula 4. The reconstruction loss δ between these two full-text vectors can also be calculated as 6.5 based on Formula 5 and Formula 6. We add S_2 and S_3 into the candidate set, and find that the value of δ is 2.5 and 4.5 when the candidate sets are (S_1, S_2) and (S_1, S_3), respectively. Although the amount of information in S_2 and S_3 is the same, adding S_3 results in a higher reconstruction loss compares to S_2. This is because, despite the fact that the feature represented by the first dimension of the vector is more important, the selected sentence set (S_1) already contain some of this feature, and continuing to select sentences containing this feature will lead to redundancy. In this case, BugSum tends to select sentences that contain other features to maintain the comprehensiveness of the selected sentence set.

The process of the beam search algorithm is illustrated in Algorithm 1. We use L_{new} and L_{old} to store the candidate sentence sets for the current iteration and next iteration, respectively. In each iteration, for each candidate sentence set l_i in L_{old}, a new sentence is added to form a new candidate set l_{new}. If l_{new} can be further extended under the word amount limitation θ, it will be added into L_{new}. Otherwise, it will be added into L_{Chosen} as one of the promising sentence sets used to form the summary. L_{new} and L_{Chosen} are maintained to retain b sentence sets with the highest δ. b is the beam size of the beam search algorithm. After all iterations are complete, the sentence set with the highest δ will be selected to form the summary. We denote this sentence set as *Chosen*. Finally, BugSum concatenates the sentences from *Chosen* in their original order in the bug report to obtain the summary *i.e.*, *SUM*.

4 EXPERIMENTS

We conduct experiments to evaluate our approach by answering the following research questions:

- **RQ1:** How does BugSum perform against baseline approaches?
- **RQ2:** To what extent does BugSum reduce the controversial sentences being selected into the summary?
- **RQ3:** How do the parameters influence the performance of BugSum?
- **RQ4:** How do sentence feature extraction and dynamic selection influence the performance of BugSum?

4.1 Experimental Setup

We implement BugSum on PyTorch [38]. All experiments are deployed on a single machine with the Ubuntu 16.04 operating system, the Intel Core (TM) i7-8700K CPU, the GTX1080ti GPU, and 16 GB memory.

4.1.1 Datasets. We design our experiments on two popular benchmark datasets, *i.e.*, *Summary Dataset* (SDS) [43] and the *Authorship Dataset* (ADS) [17], which consist of 36 and 96 bug reports, respectively. Each bug report is annotated by three annotators to ensure quality. The annotators were asked to write an abstractive summary (*AbsSum*) in around 25% of the length of the bug report using their own words. They were also asked to list the sentences from the original report that gave them the most information when writing the summary. For each bug report, the sentences listed by more than two annotators are referred to as the golden standard sentences set (*GSS*) [43].

4.1.2 Baseline Approaches. We reproduce eight previous methods to compare with our approach.

DeepSum [22] is an unsupervised approach for bug report summarization that focuses on predefined field words and sentence types. Centroid [40], MMR [6], Grasshopper [52], and DivRank [30] are unsupervised approaches for natural language summarization. They are enhanced by Noise Reducer [28] and implemented for bug report summarization. We use the enhanced version of these four approaches in our experiments. Hurried [27] is an unsupervised approach that imitates human reading patterns, connects sentences based on their similarity, and chooses sentences with the highest possibility of being read during a random scan. DeepSum and Centroid mainly rely on word frequency in bug reports. MMR selects sentences based on their novelties. Grasshopper, DivRank, and Hurried focus on context information. It should be noted here that the context information not only contains evaluation behaviors used in our approaches, but also the relationships formed by sentences similarities.

BRC [43]and ACS [17] are supervised approaches for bug report summarization that use annotated bug reports as the training data for their classifiers. They score and choose sentences base on the classifiers. Due to the lack of annotated data, we use leave-one-out [43] procedure in our experiments. The leave-one-out procedure randomly chooses one bug report as the test set and the rest as the training set. We repeat this procedure ten times and use the average value as the final result.

4.1.3 Evaluation Metrics. We evaluate the performance of approaches from the perspective of accuracy and readability. The *Precision*, *Recall*, *F-Score*, and *Pyramid* metrics are used to measure the accuracy of the approaches, and the readability of approaches are measured in the form of the *Rouge-1* and *Rouge-2* metrics.

We use the *Precision*, *Recall*, and *F-Score* metrics, which are calculated from the selected sentence set *Chosen* and the golden standard sentence set *GSS*, to measure the accuracy of the summaries. Given a selected sentence set *Chosen* and the corresponding summary *SUM*, these metrics are calculated as follows:

$$Precision = \frac{|Chosen \cap GSS|}{|Chosen|} \tag{7}$$

$$Recall = \frac{|Chosen \cap GSS|}{|GSS|} \tag{8}$$

$$F{-}score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{9}$$

Pyramid [36] precision is proposed to better measure the quality of the summary when multiple annotators exist. The assessment based on *Pyramid* assumes that, sentences listed by more annotators should be preferred, with the achievement of a certain accuracy.

$$Pyramid = \frac{Num_{ChosenListed}}{Num_{MaxListed}} \tag{10}$$

$Num_{ChosenListed}$ is the amount of times that the sentences in *Chosen* are listed by annotators, while $Num_{MaxListed}$ is the maximum possible amount for the corresponding word amount limitation. For example, three sentences are referenced by 2, 3, and 3 annotators, respectively. When two sentences are required to form the summary, selecting the last two sentences can result in a maximum $Num_{MaxListed}$ of 6. If in fact, we choose the first two sentences, the value of $Num_{ChosenListed}$ is 5. Therefore, the *Pyramid* of this selection can be calculated as $\frac{5}{6}$ according to Formula 10.

The *ROUGE* toolkit [23] measures a method's qualities by counting continuously overlapping units between the summary *SUM* and the ground truth *AbsSum*. For each bug report, we calculate the *Rouge-n* value with all three *AbsSum* written by the three annotators, and use their average value as the final *Rouge-n* score. *Rouge-1* and *Rouge-2* are used in our experiments due to their abilities in human-automatic comparisons [37].

$$Rouge{-}n = \frac{\sum_{s \in AbsSum} \sum_{n-gram \in s} Count_{match}(n{-}gram)}{\sum_{s \in AbsSum} \sum_{n-gram \in s} Count(n{-}gram)} \tag{11}$$

In Formula 11, n is the n-gram length. The numerator is the number of n-gram overlapping units between *SUM* and *AbsSum*, while the denominator is the number of n-gram in *AbsSum*.

4.2 Answer to RQ1: Overall Performance

We compare the performance of BugSum with 8 baselines as introduced in Section 4.1.2. We use the average of 10 times experiments as the final results. Table 1 and Table 2 show the overall performance of BugSum against eight baselines over SDS and ADS, respectively. A gray cell represents BugSum outperforming a baseline approach with $p{-}value < 0.05$ by the paired Wilcoxon signed rank test [15]. Experiment results show that, BugSum outperforms baseline approaches on almost all metrics and reaches the second place in the metrics *Precision* and *Pyramid* over SDS.

The *Recall* of BugSum is significantly higher than that of comparative approaches, and the reason may be that: the *Recall* reveals the coverage of salient sentences. Due to the redundancy in sentences, similar sentences tend to be scored with close scores. Therefore, whenever a salient sentence is selected, previous approaches may also select sentences that contain redundant features of this salient sentence, which leads to the drop in *Precision*. The coverage of salient sentences has to be decreased to maintain relatively high *Precision*. BugSum selects sentences while also considering their contributions to the comprehensiveness of selected sentences, which can prevent part of the noise sentences from being selected. This makes BugSum has high *Recall* while maintaining relatively high *Precision*. Approaches such as Hurried, Grasshopper, and DivRank rely on context information, they use sentence similarity as one of the criteria for constructing context information. This criterion causes bias introduced by the redundancy in sentences to have a greater impact on these approaches, which makes them have relatively low *Recall* with the similar *Precision*. By contrast, MMR selects sentences based on their novelties, which makes it has relatively high *Recall* while having similar *Precision* over ADS. DeepSum also has relatively high *Recall*, as it re-initiates similar sentences during its pre-processing step.

Table 1: Overall Performance on SDS.

	F-score	Precision	Recall	Pyramid	R-1	R-2
Centroid	0.343	0.536	0.270	0.460	0.472	0.126
Grasshopper	0.369	0.527	0.301	0.523	0.509	0.135
DivRank	0.378	0.590	0.301	0.545	0.527	0.138
ACS	0.397	0.596	0.335	0.600	0.515	0.134
BRC	0.401	0.572	0.351	0.629	0.522	0.140
Hurried	0.410	**0.711**	0.300	**0.710**	0.527	0.153
MMR	0.429	0.617	0.353	0.551	0.498	0.145
DeepSum	0.462	0.621	0.388	0.624	0.563	0.177
BugSum	**0.493**	0.629	**0.413**	0.661	**0.589**	**0.194**

Table 2: Overall Performance on ADS.

	F-score	Precision	Recall	Pyramid	R-1	R-2
DivRank	0.325	0.446	0.282	0.542	0.499	0.201
Centroid	0.337	0.488	0.280	0.561	0.473	0.183
MMR	0.396	0.505	0.356	0.585	0.503	0.206
Grasshopper	0.361	0.445	0.337	0.546	0.503	0.200
BRC	0.411	0.566	0.349	0.656	0.516	0.206
Hurried	0.417	0.576	0.346	0.635	0.540	0.239
ACS	0.453	0.609	0.396	0.672	0.546	0.231
DeepSum	0.457	0.606	0.394	0.681	0.553	0.249
BugSum	**0.491**	**0.611**	**0.417**	**0.692**	**0.564**	**0.270**

The results of the *Pyramid* metric show a similar trend with *Precision*. BugSum performs smoothly on both datasets, achieving the second highest and highest performance over SDS and ADS, respectively.

Readability is assessed using the *Rouge-n* score. The results suggest that the summaries generated by BugSum are more readable than all baseline approaches.

The characteristics of datasets can significantly affect the performance of different approaches. For example, ACS is based on authorship. ACS uses bug reports posted by the same author as the training set to train a sentence classifier. The bug reports in ADS have this kind of authorship, which make ACS has relatively high performance on the ADS dataset. We find that approaches

based on context information, such as MMR, DivRank, and Hurried, exhibit a significant performance drop when testing over ADS. To understand the cause of this performance drop, we count the number of sentences and the proportion of sentences related to the evaluation behaviors in SDS and ADS, respectively. We find that bug reports in ADS only contain an average of 39 sentences. Compared with an average of 65 sentences in SDS, ADS has a relatively small amount of sentences, which makes the sentences in the description more important. In SDS and ADS, 44.5% and 30.7% of sentences respectively are influenced by evaluation behaviors. This indicates that there are relatively fewer evaluation behaviors in ADS, which results in a performance drop for approaches that rely on context information. Despite this, however, BugSum still achieves the state-of-the-art performance in ADS. The reason is that, BugSum only uses evaluation behaviors to emphasize the believability of sentences, but does not entirely rely on them.

Result 1: BugSum outperforms baseline approaches on most metrics over these two datasets. The improvement in terms of *F-score* and *Rough-1* is up to 0.166 and 0.117, respectively. In particular, the *Recall* of BugSum outperforms baseline approaches by up to 0.143. This means that BugSum can cover more salient sentences by reducing semantic redundancy while also maintaining comparatively high accuracy.

4.3 Answer to RQ2: Controversial Sentence Reduction

As was introduced in Section 2.1, the information contained in controversial sentences is likely to be incorrect. Therefore, selecting these sentences into summaries may introduce misleading information. BugSum evaluates the believability of sentences and aims to reduce the possibilities of controversial sentences being selected into summaries. In order to determine the extent to which BugSum reduces these possibilities, we first need to identify which controversial sentences are contained in our datasets ADS and SDS. In other words, we have to build a controversial sentence set as the ground truth. To ensure correctness, we only choose sentences that are explicitly disapproved by all evaluations, where the information in the sentence is also manually confirmed to be incorrect. We recruit five experienced programmers, who have at least four years of programming experience. They determine whether a sentence is controversial based on the following criteria:

- The sentence should be selected by at least one baseline approach.
- The sentence should have been explicitly evaluated by at least one sentence, and all of these evaluation sentences should express negative opinions.

We select sentences that are determined to be controversial by all five programmers. We obtain 7 and 16 controversial sentences from SDS and ADS, respectively. For each baseline and BugSum, we check the total number of controversial sentences that have been selected into the summaries over ADS and SDS.

As shown in Fig. 3, BugSum only select 8.7% of controversial sentences into the summaries, which reduces the controversial

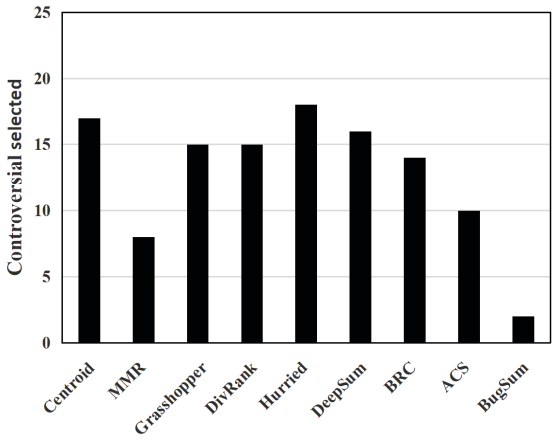


Figure 3: Selected controversial sentences.

sentences in summaries by up to 69.6% compared to baseline approaches. We also observe that approaches like Grasshopper, DivRank, and Hurried based on context information, and approaches such as DeepSum and Centroid based on word frequency select more controversial sentences. This validates our assumption proposed in Section. 2.1. The controversial sentences are discussed by a series of comments before they are disapproved. Words or sentences related to the controversial sentences will appear more times in bug reports. Thus, approaches based on word frequency or context information are likely to select more controversial sentences.

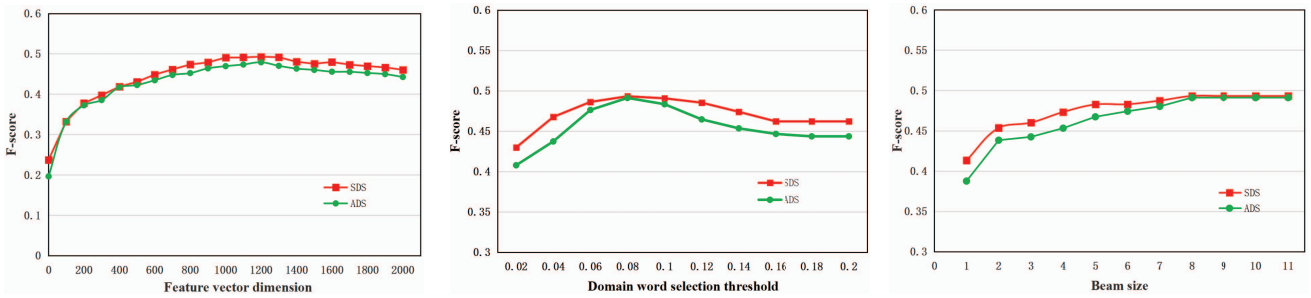
Result 2: Controversial sentences are likely to be selected by the baseline approaches. BugSum can significantly reduce the possibility of controversial sentences being selected into the summary by up to 69.6% according to our careful empirical evaluation.

4.4 Answer to RQ3: Influence of Parameters

BugSum contains three parameters: feature vector dimension, domain word selection threshold, and the beam size of the beam search algorithm. To find out how these parameters influence the performance of BugSum, we perform the following experiments.

4.4.1 Feature Vector Dimension. BugSum uses sentence vectors and a full-text vector to represent important information in bug reports. The dimension of these feature vectors may affect the performance of BugSum. We evaluate the performance of BugSum with the vector dimension from 1 to 2000. In Fig. 4(a), we present the *F-score* values of BugSum.

The performance curves of BugSum on SDS and ADS exhibit a similar trend. The performance of BugSum declines rapidly when the dimension of feature vectors is lower than 200. It grows steadily when the dimension is between 200 and 1000. When the dimension is between 1000 and 1400, the performance of BugSum remains stable and peaks when the dimension reaches 1200. The performance begins to decrease when the dimension exceeds 1400. The reason for this is that a low-dimension feature vector can only retain limited features with insufficient information, which can lead to worse



(a) (b) (c)

Figure 4: Performance of BugSum influnced by different parameters.

performance. By contrast, when the dimension is too large, noisy features are also included in the feature vectors, which causes performance degradation.

We have also checked the performance of BugSum in terms of other metrics, and obtained quite similar results. Thus, we set the dimension of feature vectors to 1200 in all our experiments, as at around this value, the performance of BugSum reaches the peak on both ADS and SDS.

4.4.2 Domain Word Selection Threshold. As noted in Section 3.3.1, we build the connection between the description and comments based on the sharing of domain words to assess the believability of sentences in the description. When selecting domain words, we need to set the threshold θ to the TF-IDF value. In this experiment, we test the sensibility of θ, from 0.02 to 0.20.

As can be seen in Fig. 4(b), the performance of BugSum increases rapidly over two datasets when θ grows from 0.02 to 0.06. Subsequently, as the value increases from 0.06 to 0.1, we obtain comparative performance. When θ is higher than 0.1, the performance of BugSum first declines slightly and then remains stable from the point at around 0.18. The reason is that, when θ is too small, the number of selected domain words will be large. Many links, including noises, may be constructed between the description and comments, which causes a further performance drop. On the contrary, when θ is too high, few domain words can be selected, meaning that only a very limited amount of links can be built. The input information for BugSum is not rich enough, so its performance also drops. When θ is higher than a certain value, such as 0.18 in Fig. 4(b), the amount of domain words is too small, and the relation between the description and comments can no longer affect the selection. Therefore, the performance becomes stable again. We also observe that the performance over ADS is more sensitive to the change of θ. This is because there are fewer sentences in ADS than in SDS, so the sentences in the description play a more critical role in ADS. The noises introduced by θ will have more effects on ADS than on SDS. We also check the performance using other metrics and obtain similar results. Overall, we set the threshold of domain word selection to 0.08 in all our experiments.

4.4.3 Beam Size. BugSum generates a summary based on the beam search algorithm. As introduced in Section 3.4, the beam search algorithm maintains b candidate sentence sets. In Fig. 4(c), we illustrate the performance of BugSum in the form of the *F-score* metric, when b is between 1 and 11.

The performance of BugSum increases along with b until it reaches the value of 8, after which the performance becomes stable. Additional growth of the beam size cannot improve the performance of BugSum. The computational complexity of the search algorithm increases significantly as the beam size increases. Thus, we set the beam size to 8 in all our experiments to balance the performance of BugSum and the computational time consumption.

> **Result 3**: The dimension of the feature vector seriously affects the performance of BugSum. The threshold of domain word selection and the beam size also have a noticeable effect on the performance of BugSum. BugSum can achieve its highest performance by setting these parameters appropriately.

4.5 Answer to RQ4: Ablation Study

In our approach, we implement the Sentence Feature Extraction (SFE) to extract textual features from sentences, and Dynamic Selection (DS) to improve the comprehensiveness of the chosen sentences. We deploy an ablation study to test the effectiveness of these two components against the commonly used alternative strategies.

Bag-of-Words (BoW) is one of the most popular representation strategies [50], which preserves the word frequency and ignores the original order or relationship between neighboring words. The sentence score method has been commonly used in previous approaches [6, 22, 27, 30, 40, 52], which we denote as SSM. SSM selects sentences with the highest score under the word amount limitation. In this experiment, SSM uses the cosine similarity between the sentence vector S_i and the full-text vector DF as the score of sentence i. We use BoW as an alternative strategy for SFE and SSM as a replacement for DS.

We illustrate the performance of the model under different combinations of alternative strategies in Table 3. We find that the replacement of any strategies will lead to a significant drop in most metrics. The replacement of the sentence feature extraction strategy significantly impacts BugSum's *Precision*, *R-1*, and *R-2*. The reason is that, the domain textual features in the sentences include word frequency and word context. The BoW strategy can only preserve word frequency information, which leads to a performance

Table 3: Performance Using Different Strategies.

Dataset	Strategy	F-score	Precision	Recall	Pyramid	R-1	R-2
SDS	BoW SSM	0.297	0.411	0.241	0.307	0.436	0.092
	BoW DS	0.396	0.496	0.344	0.550	0.517	0.125
	SFE SSM	0.381	0.522	0.306	0.543	0.509	0.113
	SFE DS	**0.493**	**0.629**	**0.413**	**0.661**	**0.589**	**0.194**
ADS	BoW SSM	0.294	0.405	0.254	0.493	0.460	0.112
	BoW DS	0.386	0.451	0.353	0.562	0.512	0.207
	SFE SSM	0.377	0.467	0.322	0.528	0.487	0.180
	SFE DS	**0.491**	**0.611**	**0.417**	**0.692**	**0.564**	**0.270**

drop, especially in terms of precision and readability. This also indicates that our approach can preserve the domain textual features in sentences. We also find that summary selection strategies heavily influence BugSum's *Recall*, a result that is caused by the redundancy in sentences. Dynamic selection, as evaluated in Section 4.3, can select sentences while considering the comprehensiveness of the selected sentences. Alternative strategies like SSM tend to select sentences with redundant semantic features, and further cause relative low *Recall* while achieving similar *Precision*.

Result 4: BugSum's sentence feature extraction strategy and dynamic selection strategy outperform alternative strategies (*i.e.*, BoW strategy and SSM strategy) in terms of 6 metrics over the datasets.

5 RELATED WORK

5.1 Bug Report Summarization

Bug report summarization, which is considered to be a promising way to reduce human effort, involves composing a summary by picking out salient sentences from the bug report. Rastkar et al. [43] and Jiang et al. [17] extracted sentences based on feature classifiers that were trained using manually annotated bug reports. The performance of feature classifiers relies heavily on the quality of the training corpus [27], which requires the annotators to have certain expert knowledge and massive manual efforts. Arya et al. [1] labeled comments with their possible contained information, and let users choose corresponding sentences based on their requirements. Radev et al. [40] compressed each sentence into a vector based on their TF-IDF values, and assessed sentences based on their similarity to the average of all sentence vectors. Other approaches [30, 52] have attempted select sentences according to reference relations, which were enhanced by a noise removal strategy designed by Senthil et al. [28]. Lotufo et al. [27] scored their sentences based on imitating human reading patterns, connected sentences according to their similarities, and chose the sentences with the highest possibilities of being reached during a random traverse. Jiang et al. [22] focused on predefined field words and sentence types, and scored sentences based on the weight of words. In this paper, we have proposed a novel unsupervised algorithm for bug report summarization that can efficiently reduce the possibility of controversial sentences been selected into the summary.

5.2 Summarization of NLP

Text summarization is one of the key applications of natural language processing for information condensation [32]. Wang et al. [46] generated summaries for meeting records through templates, which required considerable manual effort to obtain. Cheng et al. [8] transformed the bug summarization into a classification task, by using LSTM as a recurrent document encoder to represent documents. Nallapati et al. [33] took the position of sentences into consideration to minimize the negative log-likelihood between the prediction and the ground truth by using an RNN based sequence model. Jadhav et al. [16] implemented the pointer network to add the salience of words into the prediction process. Narayan et al. [34] optimized the *Rouge* evaluation metric through a reinforcement learning objective. Zhou et al. [51] designed an end-to-end neural network to combine the sentence scoring process and the sentence selection process. The above approaches have accelerated the development of understanding software artifacts [35], *e.g.*, source code and bug report.

5.3 Deep Learning in Software Engineering

In recent years, deep learning has been increasingly adopted to improve the performance of software engineering tasks [48]. Moreno et al. [31] and Matskevich et al. [29] utilized neural networks for source code analysis by integrating abstract syntax trees (*i.e., AST*) and code textual information to generate comments. Similarly, Wang et al. [47] combined API sequence information with neural networks, and generated descriptions for object-related statement sequences. Moreover, Linares-Vásquez et al. [25] and Buse et al. [5] used neural networks to generate commit messages through extracting code changes. Jiang et al. [18] improved the results of neural networks by adding filters to filter out the likely poor predictions. Liu et al. [26] employed the pointer network to deal with out-of-vocabulary (i.e., OOV) words. While deep learning is an exciting new technique, it is still debatable as to whether this method can be implemented in a way that benefits SE [12, 14].

6 CONCLUSION

In this study, we present a novel unsupervised summarization approach, that considers sentence informativeness, believability and comprehensiveness, to generate more reliable and comprehensive summaries for bug reports. Compared to 8 typical baseline approaches, extensive experiments over two public datasets show that the performance of our approach reaches the state-of-the-art performance. Our approach can be applied in practice to assist with software maintenance and reuse. In particular, our method is able to prevent most controversial sentences from being selected into the summary, which point a promising direction for the further work on conversation-based text analysis.

In the future, we plan to conduct a large-scale quantitative evaluation using more OSS projects to validate the generality of our approach, as well as a careful qualitative case study designed to deeply explore more unique characteristics of bug reports that can improve our performances.

7 ACKNOWLEDGMENTS

This paper is supported by National Key Research and Development Program of China (No. 2018YFB0204301), National Natural Science Foundation (No.61672529, No.61872373 and No. 61702534).

REFERENCES

[1] Deeksha Arya, Wenting Wang, Jin LC Guo, and Jinghui Cheng. 2019. Analysis and detection of information types of open source software issue discussions. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 454–464.

[2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 308–318.

[3] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*. 27–30.

[4] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[5] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 33–42.

[6] Jaime G Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries.. In *SIGIR*, Vol. 98. 335–336.

[7] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. 2015. Developer onboarding in GitHub: the role of prior social links and language experience. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. ACM, 817–828.

[8] Jianpeng Cheng and Mirella Lapata. 2016. Neural summarization by extracting sentences and words. *arXiv preprint arXiv:1603.07252* (2016).

[9] Peter Christoffersen and Kris Jacobs. 2004. The importance of the loss function in option valuation. *Journal of Financial Economics* 72, 2 (2004), 291–318.

[10] Damian Doyle. [n.d.]. Default English stopwords list. https://www.ranks.nl/stopwords. 2017.

[11] Qiang Fan, Yue Yu, Gang Yin, Tao Wang, and Huaimin Wang. 2017. Where is the road for issue reports classification based on text mining?. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 121–130.

[12] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[13] Beate Hampe. 2002. *Superlative verbs: A corpus-based study of semantic redundancy in English verb-particle constructions*. Vol. 24. Gunter Narr Verlag.

[14] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[15] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[16] Aishwarya Jadhav and Vaibhav Rajan. 2018. Extractive summarization with swap-net: Sentences and words from alternating pointer networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 142–151.

[17] He Jiang, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren. 2017. Mining authorship characteristics in bug repositories. *Science China Information Sciences* 60, 1 (2017), 012107.

[18] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[19] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M German. 2015. Open source-style collaborative development practices in commercial projects using GitHub. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 574–585.

[20] Won Kim, Ok-Ran Jeong, and Sang-Won Lee. 2010. On social Web sites. *Information systems* 35, 2 (2010), 215–236.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[22] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 144–155.

[23] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[24] Hui Lin and Jeff Bilmes. 2010. Multi-document summarization via budgeted maximization of submodular functions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. 912–920.

[25] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

[26] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 299–309.

[27] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. 2015. Modelling the 'hurried'bug report reading process to summarize bug reports. *Empirical Software Engineering* 20, 2 (2015), 516–548.

[28] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 11.

[29] Sergey Matskevich and Colin S Gordon. 2018. Generating comments from source code with CCGs. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*. 26–29.

[30] Qiaozhu Mei, Jian Guo, and Dragomir Radev. 2010. Divrank: the interplay of prestige and diversity in information networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. Acm, 1009–1018.

[31] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[32] Nikita Munot and Sharvari S Govilkar. 2014. Comparative study of text summarization methods. *International Journal of Computer Applications* 102, 12 (2014).

[33] Ramesh Nallapati, Feifei Zhai, and Bowen Zhou. 2017. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[34] Shashi Narayan, Shay B Cohen, and Mirella Lapata. 2018. Ranking sentences for extractive summarization with reinforcement learning. *arXiv preprint arXiv:1802.08636* (2018).

[35] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology* 31, 5 (2016), 883–909.

[36] Ani Nenkova, Rebecca Passonneau, and Kathleen McKeown. 2007. The pyramid method: Incorporating human content selection variation in summarization evaluation. *ACM Transactions on Speech and Language Processing (TSLP)* 4, 2 (2007), 4.

[37] Karolina Owczarzak, John M Conroy, Hoa Trang Dang, and Ani Nenkova. 2012. An assessment of the accuracy of automatic evaluation in summarization. In *Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization*. Association for Computational Linguistics, 1–9.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[39] Martin F Porter. 1980. An algorithm for suffix stripping. *Program* 14, 3 (1980), 130–137.

[40] Dragomir R Radev, Hongyan Jing, Małgorzata Styś, and Daniel Tam. 2004. Centroid-based summarization of multiple documents. *Information Processing & Management* 40, 6 (2004), 919–938.

[41] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Piscataway, NJ, 133–142.

[42] Rastkar, Sarah, Murphy, C Gail, Murray, and Gabriel. 2010. Summarizing software artifacts: a case study of bug reports. (2010).

[43] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering* 40, 4 (2014), 366–380.

[44] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

[45] Vladimir N. Vapnik. 2000. *The Nature of Statistical Learning Theory*. Springer,.

[46] Lu Wang and Claire Cardie. 2013. Domain-independent abstract generation for focused meeting summarization. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 1395–1405.

[47] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. 2017. Automatically generating natural language descriptions for object-related statement sequences. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 205–216.

[48] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 334–345.

[49] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2014. Towards effective bug triage with software data reduction techniques. *IEEE transactions on knowledge and data engineering* 27, 1 (2014), 264–280.

[50] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics* 1, 1-4 (2010), 43–52.

[51] Qingyu Zhou, Nan Yang, Furu Wei, Shaohan Huang, Ming Zhou, and Tiejun Zhao. 2018. Neural document summarization by jointly learning to score and select sentences. *arXiv preprint arXiv:1807.02305* (2018).

[52] Xiaojin Zhu, Andrew Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving diversity in ranking using absorbing random walks. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*. 97–104.

[53] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.