Empirical Software Engineering (2021) 26: 119
https://doi.org/10.1007/s10664-021-10008-2

How to cherry pick the bug report for better summarization?

Haoran Liu[1] · Yue Yu[1] · Shanshan Li[1] · Mingyang Geng[1] · Xiaoguang Mao[1] · Xiangke Liao[1]

Accepted: 15 June 2021 / Published online: 3 September 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

Bug reports, as a frequently consulted software asset, are maintained and evolved in software communities. A large number of bug reports with complex discussions are accumulated during the software evolution. It has been proven that an accurate and concise summary can help developers reduce the time effort spent going through the entire content of bug reports. Prior works select salient sentences that contain the most semantic information to form summaries. Their performance is limited due to the lack of consideration of controversial standpoints among developers' comments and the redundancy in sentences. In this paper, we study the possibility of assessing comments' opinions from discussions, and which kind of sentences are more likely to have redundant information. Based on these studies, we propose two new factors, ***Believability*** and ***Informativeness***. The former measures the degree of approved or disapproved to a sentence within discussions, and the latter assesses the amount of information contained in the summary. Accordingly, we design ***BugSum***, a supervised approach to generate summaries with a two-phase method. In the measuring phase, we propose a classification method that combines the advantages of Deep Pyramid CNN and Random Forest to assess the believability of sentences in bug reports. In the selection phase, BugSum integrates an auto-encoder network for semantic feature extraction with the believability of sentences, and optimizes the informativeness of generated summaries through a dynamic selection of salient sentences. Extensive experiments show that our approach outperforms 8 comparative approaches over two public datasets and one customized dataset. In particular, the probability of adding controversial sentences that are clearly disapproved by other developers into the summary is reduced by up to 64.7%.

Keywords Bug report summarization · Deep learning · Software maintenance · Mining software repositories

Communicated by: Ali Ouni, David Lo, Xin Xia, Alexander Serebrenik and Christoph Treude

This article belongs to the Topical Collection: *Recommendation Systems for Software Engineering*

✉ Yue Yu
yuyue@nudt.edu.cn

✉ Shanshan Li
shanshanli@nudt.edu.cn

[1] College of Computer Science and Technology, National University of Defense Technology Changsha, Hunan, China

1 Introduction

With the increasing recognition of open source communities, to share and improve the quality of projects and codes, many individual programmers and commercial companies (Kalliamvakou et al. 2015) tend to manage their development tasks using online bug repositories (*e.g., Bugzilla*[1]) or issue tracking systems (*e.g., Issue Tracker in GitHub*[2]). A large number of bug reports and discussions are accumulated due to the surge of open-source processes. These bug reports and discussions are regarded as valuable resources for software development (Casalnuovo et al. 2015) and long-term maintenance (Bettenburg et al. 2008).

On these online communities, a bug report is usually organized as an open post with a discussion section akin to that on social web sites (Kim et al. 2010). When finding a new issue, contributors may post it as a bug report, and describe it in the description section of the post using a natural language (Bettenburg et al. 2008; Zimmermann et al. 2010; Fan et al. 2017). Subsequently, 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 reply to those standpoints with their own opinions based on the development status, also in the form of comments. Therefore, the scale of a bug report increases rapidly through continuously iterative discussion (Rastkar et al. 2010). According to our study, 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 (Rastkar et al. 2010). 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.

Automatically generating summaries for bug reports has been proven to be a promising method (Rastkar et al. 2014) to solve this problem. Previous works are based on supervised (Rastkar et al. 2014; Jiang et al. 2017) or unsupervised (Li et al. 2018; Zhu et al. 2007; Mei et al. 2010)methods, and constitute a summary by selecting salient sentences. The performance of traditional supervised approaches relies heavily on the quality of the training corpus (Lotufo et al. 2015), which requires massive manual efforts 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, the discussions are cross-validated among different stakeholders according to their own experiences, and some comments will be disapproved by other participators; 2) a group of comments supporting the same topic has 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 informativeness due to the word length limitation.

In this paper, we conduct studies on 34,140 bug reports from 8 popular open-source projects. We find that discussions existed in 71.6% of bug reports. In these bug reports, 20.6% of sentences have explicit opinions, and 15.4% of sentences are approve or disapproved by

[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

other sentences. This study suggests that the ***evaluation behaviors*** (Lotufo et al. 2015) are pervasive in bug reports, and we may be able to assess whether sentences are believable by developers' discussions and their opinions. Our studies also show that, 78.2% of sentences that discussing the same topic share at least one topic-related word, and 17.4% of them contain similar information. It indicates that selecting sentences within the same discussion to form a summary is more likely to include redundant information.

Based on these studies, we propose two new factors, ***Believability*** and ***Informativeness***. Believability measures the degree that a sentence has been supported against disapproved among the interactive discussions, and informativeness assess the amount of information in summaries while considering the redundancy in sentences. Accordingly, we propose a supervised approach of bug report summarization, called ***BugSum***. BugSum generates summaries according to the following two phases. In phase one, Sentence believability scores are assessed by a classification model that combines the advantage of Deep Pyramid CNN (DPCNN) (Johnson and Tong 2017) and random forest (Surhone et al. 2010). In phase two, BugSum uses a trained auto-encoder network to extract semantic features by converting the sentences to vectors. For each sentence, we combine the believability score with its semantic features to reduce the possibility of controversial sentences being selected into the summary. The informativeness of generated summaries is optimized using a dynamic selection strategy.

We compared ***BugSum*** with eight comparative summarization approaches over two public datasets and one dataset that we manually constructed. Experimental results demonstrate that our approach outperforms comparative approaches in terms of metrics that have been widely used in previous studies.

The main contributions of this paper include:

- We design two novel factors, *i.e.*, believability and informativeness. For believability, a combination of convolutional neural networks and traditional classification algorithms are applied to optimize the accuracy.
- We propose a supervised bug summarization approach BugSum by integrating the auto-encoder network, sentence believability assessment and dynamic selection effectively. To facilitate the reproduction, we make the source code of BugSum public available[4]
- The probability of adding controversial sentences, which are clearly disapproved by other developers during discussion, into the summary is reduced by up to 64.7% according to our careful manual evaluation.
- We construct a dataset contains 39 annotated bug reports and 62 controversial sentences. It can be used to evaluate the model's ability of handling controversial sentences. To facilitate the community, we make this new dataset public available[5].
- We design extensive evaluations on two public datasets and one our manually constructed dataset to demonstrate that our approach achieves the state-of-the-art performance. Our approach outperforms the state-of-the-art approach by 8.1% and 6.7% in terms of F-score and Rouge-1 metrics, respectively.

This paper is an extension of our ICPC 2020 paper "BugSum: Deep Context Understanding for Bug Report Summarization". Compared with the prior work, our new contributions include mitigation of two critical threats from the prior work, improving the model performance from two perspectives, and more detailed experiments and discussions.

[4]https://github.com/HaoranLiu14/BugSum

[5]https://github.com/HaoranLiu14/Controversial-Dataset

First, to alleviate the vagueness of the definition and assessment of summaries' informativeness, we propose a novel method to evaluate and enhance the informativeness of generated summaries (Section 3.5). While helping to better understand the factor of informativeness, this method can improve the recall of the summary generation method by up to 2.4% while maintaining similar accuracy (Section 4.4). Second, we expand our dataset by manually constructing 39 annotated bug reports from Launchpad (Section 4.1.1) to mitigate the threat posed by the lack of data on the evaluation of model performance, especially of the ability to handle controversial sentences. Third, in order to evaluate the sentence believability in the bug report more accurately, we improve our evaluation identify method by taking ambiguous replies into consideration based on the empirical rules extracted from developers' behaviors (Section 3.2.1). Experimental results in Section 4.3 show that our model can be improved by 14.1% in the aspect of handling controversial sentences. Fourth, to further improve the performance of BugSum, we design a novel classification model using the combination of convolutional neural networks and the traditional classification algorithm in sentence opinion assessment (Section 3.2.2). This method improves the accuracy of sentence opinion classification by 10.1%, and improves the BugSum's ability of handling controversial sentences by 15.3% (Section 4.5). Finally, we carry out more comprehensive experiments and analyses in this paper, including BugSum's performance, the impacts brought by the classification model (Sections 4.2, 4.3, and 4.5), and the influence of factor believability and informativeness (Sections 4.4.1 and 4.4.2). All experiment data has been revised accordingly.

The rest of the paper is organized as follows: Section 2 introduces some performance limitation prone observations about bug report. Section 3 shows the overall design of BugSum and implementation details. Section 4 experiments the performance of BugSum based on five research questions. Related works are introduced in Section 6. Lastly, we conclude our work in Section 7.

2 Motivation

In modern software development, a bug report records the discussion of software bugs according to the following process. A contributor first describes the ***issue*** and submits it as the ***description*** part of a bug report using a natural language. Subsequently, other stakeholders, including project managers, maintainers, or external contributors, discuss the reproduction, location, and possible solutions of the proposed issue in the form of comments. These valuable information make bug reports a frequently accessed resource for both software developers and researchers. According to our statistics, 23,650 comments out of 34,140 bug reports references other bug reports, and 7,048 of these referenced bug reports have been Closed. Meanwhile, the reading of bug reports is needed for many post-hoc tasks (Zhang et al. 2021; He et al. 2020; Han et al. 2020; Arya et al. 2019; Lotufo et al. 2015; Bishnu and Bhattacherjee 2012; John et al. 2011; Anvik et al. 2006), such as the reproduction of historical bugs or familiarization with the evolutionary history of software. Although the description of the bug report contains the observed behavior of the bug when it was discovered, some valuable information such as solutions or detailed reproduction steps are intertwined with the developer's discussion, and therefore need to be identified and summarized.

During the discussion in the bug report, 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

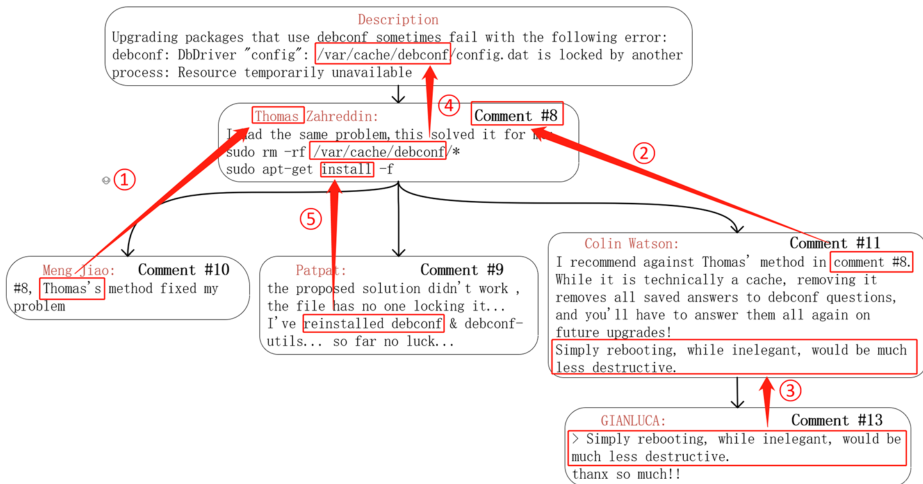

Fig. 1 Evaluation behaviors among sentences. (Bug #349467 of deconf from launchpad)

opinions (*i.e.* ***evaluation behaviors***), also in the form of comments. Such interaction structure has been referred to as conversation-based text in previous studies (Jiang et al. 2017; Rastkar et al. 2014). In this paper, we refer to a group of comments interconnected by evaluation behaviors as a ***conversation***.

In order to develop a thorough understanding of conversations in the bug report, we analyzed 34,140 bug reports from 8 popular open-source projects (6,954 Hadoop, 1,177 ZooKeeper, 5,705 Derby, 1,826 Yarn, 8,876 Flink, 3,710 HDFS, 2,907 Hive and 2,985 Launchpad). To observe the complete life-cycle information of bug reports, 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

For the purpose of selecting sentences containing more information, previous summarization approaches identify salient sentences based on word frequency (Radev et al. 2004), predefined structure (Lotufo et al. 2015) and so on. These methods assume that a sentence with more high-frequency words or a specific structure is more likely to contain more important information, while a sentence with more important information is more salient. Based on such assumptions, typical summarization approaches determine and select salient sentences to form a summary. However, in our dataset, there are 71.6% of the bug reports contain evaluation behaviors. Each bug report, on average, contains 20.6% of sentences that evaluate other sentences with an opinion of approval or disapproval, and 15.4% of sentences that are evaluated. In this paper, we refer to The extent to which a sentence is approved during the discussion in the entire bug report as **Believability**. It reflects the extent to which a statement is supported in these evaluation behaviors. It is a significant challenge to leverage the salience and believability of these highly discussed sentences.

For example, as shown in Fig. 1, the report describes a system bug reading *"Resource temporarily unavailable"* in the last sentence. Each comment is assigned a number from 1 based on the order in which it was posted (*i.e.* comment number). Subsequently, *Comment#8* proposes a solution, which is approved by *Comment#10* but disapproved by *Comment#9* and

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 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 (Radev et al. 2004), which would introduce a significant bias into the summary.

Therefore, when attempting to generate a high-quality summary, the salience of a candidate sentence should be determined not only by the information it contains, but also by the extent to which the standpoint is believable.

2.2 Redundancy in Sentences

As a conversation-based text, bug reports contain numerous conversations that are formed based on developers' discussion (*i.e.*, evaluation behaviors). Sentences in the same conversation that discuss relevant standpoint usually contain information of the same type, such as bug reproduction or solution discussion. A high-quality summary should contain all types of information, so that it can help stakeholders establish a comprehensive understanding of the bug report. Sentences in the same conversation, however, tend to contain redundant semantic features, such as specific words related to the discussed standpoint. Therefore, over selection of sentences of a specific information type may introduce redundancy into the summary, thereby reducing its quality.

We find that, there are 78.2% of sentences in the same conversation share at least one topic-related word, and the information contained in them is more similar. To measure the semantic similarity between these sentences, in our dataset, we vectorize sentences using the Bag-of-Words strategy, and calculate the cosine similarity between sentence vectors. The result shows that, on average, sentences in the same conversation are 17.4% more similar than sentences spread in different conversations. This means 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 Radev et al. 2004; Li et al. 2018), the other sentences in the same conversation may also have similar high weights due to the semantic similarity. Since previous approaches prefer to select sentences with high weights, under the word amount limitation, summaries generated by previous approaches may select too many sentences from the same conversation (*i.e.* sentences containing the same type of information), and may not comprehensive enough to conclude the entire bug report because of the semantic redundancy (Hampe 2002).

We refer to the extent to which a summary can contain information of all types as **Informativeness**. The summary should be generated while considering its informativeness.

3 BugSum Design

Based on the studies in Section 2, we propose two new factors. Assessing salient sentences while considering the believability of sentences can reduce the probability of selecting

controversial sentences into the summary. Meanwhile, considering the informativeness of summary during its generation can reduce the semantic redundancy in the summary.

Accordingly, we propose a supervised approach ***BugSum***. BugSum consists of two phases, the measuring phase and the selection phase. The measuring phase assesses the believability of sentences, which is based on the evaluation behaviors in bug reports and the combination of DPCNN and random forest. The selection phase extracts domain semantic features in sentences through an auto-encoder network, and uses a dynamic selection of believable sentences and informativeness enhancement to generate summaries under certain word amount limitation while considering its informativeness.

As illustrated in Fig. 2. The measuring phase includes *Bug Report Pre-processing* and *Sentence Believability Assessment*, while the selection phase includes *Sentence Feature Extraction* and *Summary Generation*. In the measuring phase, bug report pre-processing divides bug reports into sentences and removes the noises, while sentence believability assessment assigns believability scores to each sentence. In the selection phase, sentence feature extraction compresses the semantic information in sentences into sentence vectors. It uses the sentence believability scores as weights and constructs a full-text vector through weighted combining all sentence vectors. The summary generation step dynamically selects part of sentences from the bug report to reconstruct the full-text vector, and uses the sentence set with the smallest reconstruction loss as the salient sentence set. The set is expanded by the informativeness enhancement and used to form the summary.

3.1 Bug Report Pre-processing

Sentences in a bug report contain a considerable amount of noises (Xuan et al. 2014), which means that a pre-processing step for noise removal is necessary.

During the pre-processing step, a bug report is divided into sentences based on punctuation marks such as '.', '!', '?', and ';', except for when the punctuation is used as a part of a string. Function names such as "system.reboot", which includes common words "system" and "reboot", are tokenized using the software-specific regular expression (Lotufo

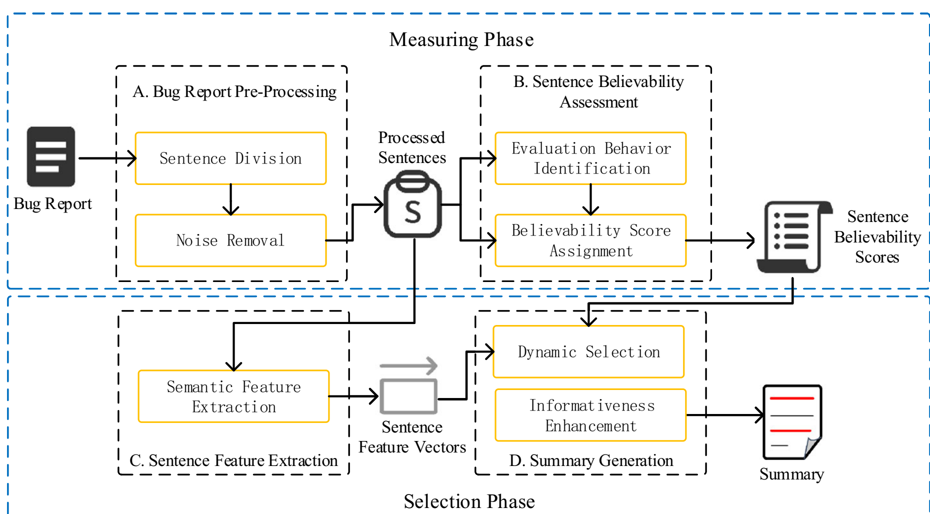

Fig. 2 The framework of BugSum

et al. 2015). Doing so can preserve the majority of the function and variable names instead of treating them as new words. BugSum further removes the stop words and stems these tokens using the poster stemmer (Porter 1980).

After this step, each bug report is split into sentences, and the word frequencies are also counted to facilitate the subsequent process.

3.2 Sentence Believability Assessment

As discussed in Section 2.1, the salience of a sentence should not only be determined by the amount of information it contains, but also by its believability. BugSum assesses sentence believability scores using evaluation behaviors, which consist of ***evaluation relationships*** (*i.e.* which sentence is the evaluator and which one is being evaluated) and ***evaluation opinions*** (*i.e.* support or disapprove). BugSum further uses the obtained believability scores as weights to construct a full-text vector, which combines the believability of sentences and domain semantic features in the entire bug report.

3.2.1 Evaluation Behavior Identification

Evaluation behaviors in bug reports have different expressions. Some evaluation behaviors are in the form of explicit replies, which are obvious and easy to be identified. Some evaluation behaviors are in the form of ambiguous replies, which do not have explicit comment targets. Evaluation behaviors related to ambiguous replies need to be speculated based on the content and context. As illustrated in Fig. 1. Arrows 1, 2, and 3 demonstrate the evaluation behaviors of explicit replies, and arrows 4 and 5 demonstrate the evaluation behaviors of ambiguous replies. Although ambiguous replies are not easy to be identified, according to our study, evaluation behaviors related to ambiguous replies affect 16.2% of comments in bug reports. BugSum identifies both types of evaluation behaviors based on their unique expressions, and uses them to identify sentence evaluation relationships.

Explicit replies often clearly indicate the author's name or the comment number of the evaluated comment, or even quote the sentences being evaluated. Quoted sentences usually have special labels (*e.g.* in Fig. 1, the symbol ">" at the beginning of the quoted sentence in comment#13). The ambiguous replies usually appear because authors think that comments they are replying to can be easily inferred, so the explicit indications are not necessary. Therefore, the ambiguous replies usually occur between comments that are close together, or between a comment and the description that are discussing the same topic. For example, as shown in Fig. 1, Comment #9 contains the sentence "the proposed solution didn't work". Since it has an explicit opinion, the "the proposed solution" should refer to one of the previous comments.

Therefore, we assume that most of ambiguous replies exist between one comment and the description, or two comments that are close to each other. These two comments should be discussing the same topic, and the later comment should have a clear opinion. Since such behaviors are prevalent during the study, based on the features above, we established three rules as follows to capture these behaviors.

- Rule 1, the comment posted later in thr bug report contains a sentence, which expresses an explicit opinion of support or disapprove.
- Rule 2, these two comments share at least one topic-related word.
- Rule 3, the distance between these two comments is within a selection distance, except for when the evaluator comment is replying to the description.

BugSum assesses words through the *TF-IDF* (Ramos et al. 2003) method, and selects words with their TF-IDF scores exceeding a threshold as topic-related words. We denote the threshold as θ_{TF-IDF}, which will be tuned in Section 4.6.2.

As for the selection distance in Rule 3, we conduct two investigations based on the dataset we mentioned in Section 2.

The first investigation is carried out on 1000 bug reports. We count the number of evaluation behaviors identified based on ambiguous replies with different selection distances, and the results are illustrated in Fig. 3a. There are a considerable amount of evaluation behaviors that can be identified by BugSum based on our rules. The number of identified evaluation behaviors decreases significantly as the selection distance increases.

We carry out the second investigation to check whether these identified behaviors actually exist.

We randomly select 100 sentences from the evaluation behaviors under different selection distance, and manually check their accuracy. The results are shown in Fig. 3b. The accuracy drops dramatically as the selection distance increases. In order to ensure the accuracy of our method, we set the selection distance to 1, under which we can still obtain many reliable evaluation behaviors.

After the above steps, we can construct the evaluation relationships in bug reports. BugSum uses an evaluation adjacency list to store the evaluation relationships. For each sentence i, BugSum stores the set of sentences that evaluates sentence i in the evaluation adjacency list, and denotes this set as $EAdj_i$.

3.2.2 Believability Score Assignment

Sentences in a bug report are supported or disapproved by other sentences during the discussion, which causes 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.

Believability is counted based on the evaluation behavior in the discussion and the opinion of the evaluator sentence. The main idea is that sentences with more support are more believable, and vice versa. The believability score of sentence i is denoted as $Bscore_i$, which is further modified base on its associated evaluation behaviors. Since multiplication is

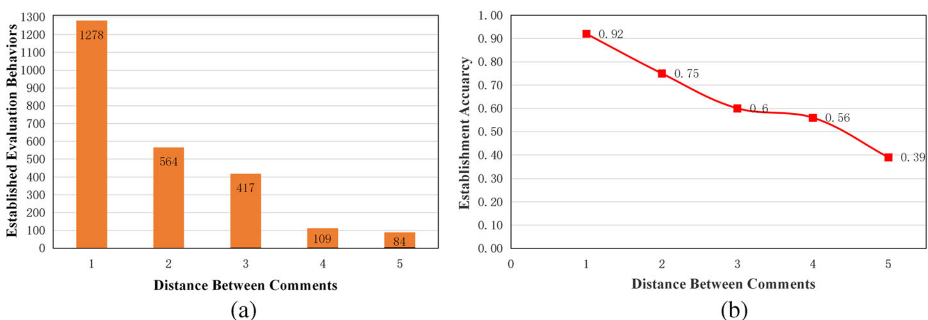

Fig. 3 Number and accuary of evaluation behaviors using different selection distance

used in the calculation of sentence believability scores, we initialize the believability score of each sentence to 1 so that it remains neutral during the calculation.

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

As we introduced in Section 3.2.1, the set of sentences evaluating sentence i is stored in an evaluation adjacency list and denoted as $EAdj_i$. We denote the size of $EAdj_i$ as $|EAdj_i|$. When $|EAdj_i| = 0$, it indicates that sentence i is not evaluated by any other sentences, so its believability score remains 1. When $|EAdj_i| > 0$, $Bscore_i$ is modified based the evaluator sentences in $EAdj_i$. For each sentence in $EAdj_i$, the weight of its influence is decided by its believability score and its opinion towards the evaluated sentence. BugSum uses opinion scores to measure the opinions of evaluator sentences. The opinion score of sentence i is denoted as $OPscore_i$, which is assigned via a pre-trained classification model. The value of $OPscore_i$ is between -1 and 1, which indicates the extent to which the evaluator sentence expresses support or disapprove. 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 an incorrect sentence is not necessarily correct.

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

Measuring the opinion score of a sentence can be regarded as a binary classification problem, positive or negative. It has been proven that convolutional neural networks (CNN) have a good performance on binary classification (LeCun et al. 1998). The convolutional neural networks, such as TextCNN (Kim 2014) and DPCNN, compress the input features through multiple layers, and use a fully connected layer to combine the compressed features to predict the probability of each category. These compressed features can be more accurate and concise than the initial input. Meanwhile, traditional classification methods, such as random forest, logistic regression (Kianifard and Kleinbaum 1995), and support vector machines (Vapnik 2000), classify inputs based on their original features. These classification methods perform more detailed use of the input features compared with the fully connected layer. To combine the advantages of these two models, BugSum uses their combination to predict sentence opinion scores.

We denote classification models that are used to assess sentence opinion scores as SO-Model. During the training process of the SO-Model of BugSum, we first train DPCNN until it achieves its maximum accuracy and remains stable. Subsequently, we take the input vectors of the fully connected layer in DPCNN, and treat them as the input of random forest. We then train the random forest until the maximum accuracy remains stable. The learning rate of the DPCNN model is set to 0.01 (Krizhevsky et al. 2012), the size of its embedding layer varies according to the feature size of BugSum, and so do the number of its filters. The parameter “min_samples_split” of the random forest is set to 5, and “n_estimators” is set to 10.

The training process of SO-Model is implemented in a dataset containing 3,000 manually labeled sentences. We recruit 5 experienced programmers, who have at least four years of programming experience. They label sentences based on whether they express negative opinions. In order to ensure the accuracy of the labeled data, we refer to the card sorting method during the process. The card sorting method recommends that all annotators should work on a certain number of data jointly first, so that all annotators can reach a consensus on the definition of every label. The five annotators first worked on 500 sentences together. After that, they worked in pairs to label the remaining sentences. Each labeled sentence is checked by at least two annotators. To ensure the balance of data with different labels, we labeling until the number of sentences of both types reaches 1,500, and use it as the training set of SO-Model.

The classification model 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.

3.3 Sentence Feature Extraction

In order to measure the informativeness of sentences, BugSum uses a trained auto-encoder network to generate sentence vectors, which are used to represent the semantic features of sentences. The auto-encoder consists of an encoder and a decoder. The encoder takes a sentence as input and encodes it into a vector, after which the sentence vector is decoded by the decoder to rebuild the original input sentence. The more consistent the input is with the output, the more precisely the vector can express the semantic features of the sentence.

In each iteration, the auto-encoder network takes one sentence as the input. The words in the sentence are first embedded into word vectors, after which the recurrent units of the encoder encode the word vectors into a hidden state sequentially. BugSum uses the last state of the encoder as the feature vector of the input sentence.

BugSum employs bidirectional GRU (Bi-GRU) (Cho et al. 2014) as the recurrent unit, which consists of a forward GRU and a backward GRU, to preserve both the forward and backward contextual features of sentences. BugSum 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.

We use 200,000 sentences processed by the pre-processing step in Section 3.1 to build the training set of the auto-encoder network. These sentences are selected from the dataset we discussed in Section 2. In the training process, we optimize the parameters of the auto-encoder network by minimizing the MSE loss (Christoffersen and Jacobs 2004) between the input sentences and corresponding output sentences. The widely used SGD optimizer (Bottou 2010) is applied to adjust the model's parameters, and the learning rate of the model is set to 0.01 (Krizhevsky et al. 2012). The training process of the model continues until the MSE loss reaches the minimum value and remains stable.

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

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

n is the number of sentences in the bug report, and S_i is the sentence vector of sentence i. DF represents the domain features of the bug report, as it combines the domain semantic features of all sentences and their believability scores.

3.4 Dynamic Selection

Algorithm 1 Beam search.

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

```
1:  L_new ← φ, L_old ← S, L_Chosen ← φ
2:  while L_old is not empty do
3:      for Each sentence set l_i in L_old do
4:          for Each sentence s_j in S do
5:              if s_j ∉ l_i then
6:                  l_new ← l_i ∪ s_i
7:                  if word amount of l_new len(l_new) < θ then
8:                      δ ← Similarity between DF~ and DF
9:                      if l_new can't be further extended then
10:                         Append l_new to L_Chosen
11:                         Update L_Chosen to reserve top-b sentences set with lowest δ
12:                     else
13:                         Append l_new to L_new
14:                         Update L_new to reserve top-b summary set with lowest δ
15:                     end if
16:                 end if
17:             end if
18:         end for
19:     end for
20:     L_old ← L_new
21:     L_new ← φ
22: end while
23: Select Chosen from L_Chosen with the lowest δ
24: return Chosen;
```

Since it is simpler and more effective to apply the extractive technique (Lotufo et al. 2015), we apply this technique for Bugsum, which selects salient sentences from the bug report to form a summary. As discussed in Section 2.2, the generated summary should include sentences with various information, so that it can help stakeholders establish a comprehensive understanding of the bug report. Due to the semantic redundancy in sentences of the same information type, a smaller semantic redundancy represents a richer information type in the summary. In other words, the informativeness of a summary during summary generation can be counted according to the semantic redundancy.

We use the feature vectors generated by the autoencoder to represent the semantic information in sentences, and count the informativeness of generated summary based on the difference (*i.e.* reconstruction loss) between the generated summary and the full-text. The smaller the difference, the less semantic redundancy in the summary, and the higher its informativeness. The set of selected sentences is denoted as $Chosen$, which contains k

sentences used to form the summary. BugSum uses the sentences in $Chosen$ 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.

$$\widetilde{DF} = \sum_{i \in Chosen} Bscore_i * S_i \tag{4}$$

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

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

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 contains less semantic redundancy and has a higher informativeness. 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 (Lin and Bilmes 2010). 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 informativeness of selected sentences during the selection process. For example, in Fig. 4, 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

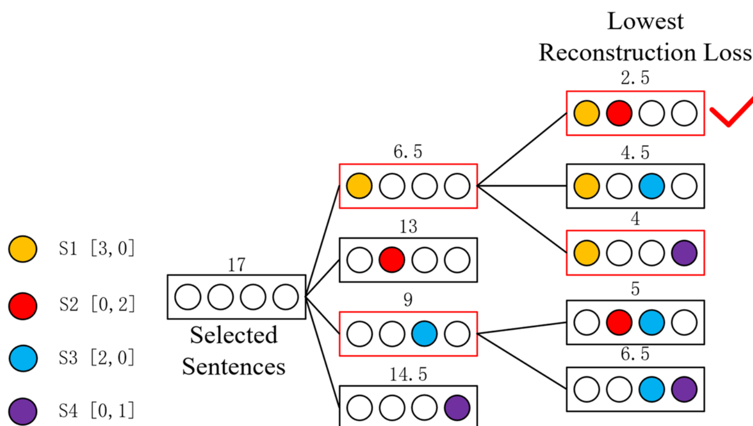

Fig. 4 An example of beam search

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 contains 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 informativeness of the selected sentence set.

The process of the beam search algorithm is illustrated in Algorithm 1. We use lists 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 list L_{old}, a new sentence is added into l_i to form a new candidate sentence set l_{new}. If l_{new} can be further extended under the word amount limitation θ, it will be added into list L_{new}. Otherwise, it will be added to list L_{Chosen} as one of the promising sentence sets used to form the summary. List 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 in list L_{Chosen} with the highest δ will be selected to form the summary. We denote this sentence set as the salient sentence set *Chosen*.

3.5 Informativeness Evaluation and Enhancement

During the dynamic selection process, we count the informativeness of a summary through the mean-square loss between feature vectors of the summary and the full-text. As introduced in Section 1, the bug report contains valuable information of various types, so the ideal summary should contain each type of information comprehensively. We assume that the proportion of each type of information in the summary should be similar to that in the full-text. If we select too many sentences representing a certain type into summary, it will lead to redundancy i.e., reduce informativeness. Therefore, we evaluate the informativeness of the summary by comparing the proportion of different information types contained between the summary and the full-text. The closer the proportion of each type of information in the summary to the full-text, the higher its informativeness remained. For example, if the proportion of sentences related to Bug Reproduction, Solution Discussion and Others in the full-text account for 30%, 30% and 40%, respectively. Then the summary with proportions of 20% ,40%, 40% can be regarded as less informative than the summary with proportions of 30% and 30%, 40%. This is because although the number of sentences related to Solution Discussion in the first summary occupies 40%, since it is also 10% higher than that in the full-text, it is likely that these additional 10% sentences will bring redundancy to the summary. Meanwhile, the first summary contains only 20% of the sentences related to Bug Reproduction, which is 10% lower than that of the full text, and could leads to a reduction in informativeness.

In order to obtain the percentage of various information types in the summary and the full-text, we need to classify the information types representing by the sentences. It has been proven that bug reproduction and solution discussion are the two types of information that stakeholders are most interest (Zhang et al. 2021; He et al. 2020; Han et al. 2020). Therefore, we divide the information in the bug report into three categories, bug reproduction, solution discussion, and others. Arya et al. (2019) uses a dataset containing 4,656 annotated sentences as training data, and classifies the information type of sentences by features such as time interval, sentence length and position using SVM and random forest algorithms. Since sentence information type classification is also a text classification task, we apply the

structure and training process of SO-Model directly to this task. We find that our method can improve the accuracy of classification from 0.69 to 0.77. By doing so, we can obtain the proportion of each information type in the full text and in the summary, and calculate the informativeness of the summary subsequently based on the difference between these proportions.

Besides being easier to understand intuitively, such a method can also help enhance the informativeness of the summary. When the proportion of a certain type of information in the summary is lower than that of the full-text, we increase it to improve the informativeness of the summary. Since we cannot reduce sentences that have been selected into the summary, we select more sentences of that type accordingly. After dynamic selection, BugSum applies this method to the generated summary for informativeness enhancement. An additional part of the sentence containing the missing information will be selected dynamically to extent the salient sentence set *Chosen*. Finally, BugSum concatenates the sentences in *Chosen* in their original order in the bug report to obtain the summary *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 factors believability and informativeness influence the performance of BugSum?
- **RQ4:** How do the different SO-Model influence the performance of BugSum?
- **RQ5:** How do the parameters influence the performance of BugSum?
- **RQ6:** How do sentence feature extraction, dynamic selection, and informativeness enhancement influence the performance of BugSum?

4.1 Experimental Setup

We implement BugSum on PyTorch (Paszke et al. 2017) . 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 three datasets, two of which are popular benchmark datasets *Summary Dataset* (SDS) (Rastkar et al. 2014) and the *Authorship Dataset* (ADS) (Jiang et al. 2017). These two datasets consist of 36 and 96 bug reports, respectively. In these datasets, each bug report is annotated by three annotators to ensure accuracy. The annotators were asked to write an abstractive summary (*AbsSum*) in around 25% of the length of the bug report using their own words, which is to ensure that annotators have carefully read the entire bug report and understand its contents. 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*) (Rastkar et al. 2014).

These datasets, however, evaluate model performance based on the accuracy of salient sentence selection. To better evaluate the model's ability in handling controversial sentences, we construct a new database *Controversial Dataset* (CDS). It consists of 39 annotated bug reports and 62 controversial sentences. These annotated bug reports were randomly selected from the 34,140 bug reports introduced in Section 2. In addition to writing the AbsSum and constructing the GSS following the requirements described above, the five annotators also need to determine the controversial sentence in the bug report. Each bug report in CDS contains at least one controversial sentence. To ensure accuracy, each controversial sentence in CDS is disapproved by all its evaluator sentences and determined by all five annotators. These annotated controversial sentences contain incorrect information, and selecting them into the summary may introduce bias. Since this dataset is constructed to evaluate the model's ability of handling controversial sentences, it requires that the annotated bug report must contain a controversial sentences. Hence during data annotation, the annotator will first annotate controversial sentences. If the bug report does not contain a controversial sentence, it will be skipped. Therefore, CDS can be used to test the model's ability of handling controversial sentences by counting the number of selected controversial sentences.

The size of the dataset has been an insurmountable limitation of the bug report summary generation work. The complexity of the bug report annotation process and the high demand on annotators make it difficult to increase the size of this dataset. This also becomes a threat to the accuracy of the experiment. In addition to being able to evaluate the ability of controversial sentence handling, the CDS proposed in this paper also expands more data and mitigates this threat.

4.1.2 Baseline Approaches

We reproduce eight previous methods to compare with our approach. DeepSum (Li et al. 2018) is an unsupervised approach for bug report summarization that focuses on predefined field words and sentence types. Centroid (Radev et al. 2004), MMR (Carbonell and Goldstein 1998), Grasshopper (Zhu et al. 2007), and DivRank (Mei et al. 2010) are unsupervised approaches for natural language summarization. They are enhanced by Noise Reducer (Mani et al. 2012) and implemented for bug report summarization. We use the enhanced version of these four approaches in our experiments. Hurried (Lotufo et al. 2015) 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 contains not only evaluation behaviors used in our approaches, but also the relationships formed by sentence similarities.

BRC (Rastkar et al. 2014)and ACS (Jiang et al. 2017) 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 the leave-one-out (Rastkar et al. 2014) procedure in our experiments. The The leave-one-out procedure randomly divides the data into several parts, and each time randomly takes one as the test set and the others as the training set. We divide the data into 10 parts, repeat this procedure 10 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 salient sentence selection and generated summary. The *Precision*, *Recall*, *F-Score*, and *Pyramid* metrics are used to measure the accuracy of the salient sentence selection, and the quality of generated summary 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 (Nenkova et al. 2007) 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, if 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 (Lin 2004) 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 (Owczarzak et al. 2012).

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

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. Tables 1, 2 and 3 show the

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	0.385	0.584	0.317	0.589	0.505	0.131
BRC	0.401	0.558	0.342	0.622	0.517	0.139
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	**0.499**	0.627	**0.437**	0.659	**0.601**	**0.197**

overall performance of BugSum against eight baselines over SDS, ADS, and CDS, respectively. A gray cell represents BugSum outperforming a baseline approach with $p-value <$ 0.05 by the paired Wilcoxon signed rank test (Holm 1979). 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 informativeness of currently 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 rules for constructing context information. This rule 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*.

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	0.409	0.563	0.347	0.651	0.513	0.205
Hurried	0.417	0.576	0.346	0.635	0.540	0.239
ACS	0.451	0.602	0.391	0.670	0.543	0.231
DeepSum	0.457	0.606	0.394	0.681	0.553	0.249
BugSum	**0.494**	**0.611**	**0.424**	**0.691**	**0.568**	**0.272**

By contrast, MMR selects sentences based on their novelties, which makes it has relatively high *Recall* while having similar *Precision* over ADS and SDS. DeepSum also has relatively high *Recall*, as it re-initiates similar sentences during its pre-processing step.

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

The quality of the generated summary is assessed using the *Rouge-n* score. The results suggest that summaries generated by BugSum having the best performance over these datasets.

We observe that, 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 makes 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, but have relatively high performance on CDS. To understand the cause of these performance fluctuations, we count the number of sentences and the proportion of sentences related to the evaluation behaviors in SDS, ADS, and CDS, respectively. We find that bug reports in ADS only contain an average of 39 sentences. Compared with an average of 65 sentences in SDS and 52 in CDS, ADS has a relatively small amount of sentences, which makes the sentences in the description more important. In SDS, ADS, and CDS, 44.5%, 30.7%, and 41.9% 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.

datasets. It outperforms the state-of-the-art approach by 7.7% and 5.2% in terms of F-score and Rouge-1 metrics, respectively. *i.e.* we achieve up to 51.4% (0.167/0.325 in ADS) and 25.3% (0.119/0.472 in SDS) improvements in terms of *F-score* and *Rough-1*, respectively. In particular, the *Recall* of BugSum outperforms baseline approaches by up to 53.3% (0.144/0.270 in SDS). 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. 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 and are also manually confirmed to be incorrect. The 5 annotators mentioned

Table 3 Overall performance on CDS

	F-score	Precision	Recall	Pyramid	R-1	R-2
DivRank	0.343	0.451	0.302	0.548	0.501	0.201
Centroid	0.346	0.490	0.294	0.561	0.475	0.183
Grasshopper	0.369	0.447	0.348	0.549	0.505	0.200
MMR	0.399	0.501	0.363	0.583	0.499	0.202
ACS	0.409	0.584	0.369	0.642	0.554	0.238
BRC	0.421	0.565	0.358	0.657	0.517	0.206
Hurried	0.446	0.602	0.354	0.679	0.509	0.203
DeepSum	0.467	0.609	0.406	0.685	0.554	0.249
BugSum	**0.496**	**0.613**	**0.437**	**0.687**	**0.581**	**0.253**

in Section 3.2.2 are also recruited for this task. They determine whether a sentence is controversial based on the following rules:

- 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. As noted in Section 4.1.1, in order to have a more accurate test of the methods' ability to process controversial sentences, we construct a new database CDS containing 62 controversial sentences. In the CDS, each bug report contains at least one controversial sentence. For each baseline approach and BugSum, we check the number of controversial sentences that have been selected into the summaries over ADS, SDS, and CDS, respectively. In addition, to understand the improvement brought by considering ambiguous replies, we also test the performance of BugSum considering only explicit replies, which is denoted as BugSum-E.

As shown in Fig. 5, BugSum only selects 9.4% of controversial sentences (8 out of 85) into the summaries, which reduces the controversial sentences in summaries by up to 64.7% (55 out of 85) compared to baseline approaches. Meanwhile, compared with BugSum-E, the reduction of controversial sentences is increased from 76.5% (65 out of 85) to 90.6% (77 out of 85), which indicates that the consideration of ambiguous replies can improve the deduction of controversial sentences. 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.

In order to better understand the limitations of BugSum's ability of handling controversial sentences, we perform a case study of the 8 controversial sentences that selected by BugSum. We found that these sentences are from 7 comments of 7 bug reports. We selected three of the representative sentences and their evaluator sentences in this study. As illustrate in Table 4, these sentences are evaluated by only one sentence, and the opinion of the evaluating sentence is relatively gentle. This results in these evaluated sentences having believability scores between 0.3 and 0.6. Also, these sentences contain a large number

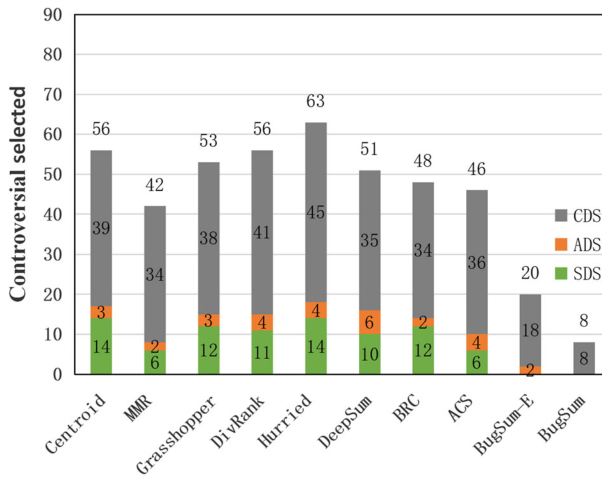

Fig. 5 Selected controversial sentences

of topic-related words, such as “drivers”, “version”, and “depends” in Sentence 1, making the sentence textually informative. Therefore, in the trade-off between textual informative and believability, BugSum decides to select these sentences into the summary. This result suggests that although our approach can significantly reduce the number of controversial sentences in the summary, there is still great potential in the design of our methods for sentence opinion assessment and sentence believability assignment.

Result 2: Controversial sentences are likely to be selected by the baseline approaches, and considering ambiguous replies can improve the reduction of controversial sentences by 14.1%. BugSum can significantly reduce the possibility of controversial sentences being selected into the summary by up to 64.7% according to our empirical evaluation.

Table 4 Example of controversial sentences selected by BugSum

	Controversial Sentence	Evaluator Sentence
1	The drivers just have to work with the version we have, and Depends does not solve anything.	If some driver happens to have issues, it needs patching.
2	I briefly reviewed the debdiff and it seems that the script will hit ‘cantcreatebase’ if the directory does exist.	I thought that too, but then both experimentation and the mkdir(1) manpage showed that mkdir -p won’t return an error if the directory already exists (it will for other problems, such as EPERM).
3	Of course, then we’d need to find a way to avoid mailing him when the 5 minutes batch time are up.	Why not special-case the action instead?

4.4 Answer to RQ3: Influence of Believability and Informativeness

In this paper, we propose two new factors: **Believability** and **Informativeness**. The experiments above show that using these two factors can help generate summaries with higher accuracy and fewer controversial sentences. In this section, we discuss how these two factors can help improve the performance of the bug report summarization.

4.4.1 Influence of Believability

Believability refers to the degree that a sentence has been supported against disapproved among the discussions in the bug report. The purpose of introducing this factor is to assess whether a sentence is believable based on relevant evaluation behaviors and sentence opinions in the bug report. As introduced in Formula 3 in Section 3.3, the sentence believability scores reduce the impact of the potentially incorrect sentences in the full-text through the combination of as weights with a vector of sentence features.

In this section, we illustrate how the factor believability affects the performance of BugSum through ablation study (introduced in Section 4.7). We add two control models, *i.e.* the model does not consider the opinion of the evaluator sentences when assessing believability (BugSum-NOp), and the model does not consider the sentence believability (BugSum-NBeli). For model BugSum-NOp, it assumes that all sentences express positive opinions and their opinion scores are set to 1. Meanwhile, BugSum-NBeli assumes that all sentences have the same believability. According to Formula 3, we set the believability score of each sentence to 1 so that each sentence contributes equally to the generation of the full-text vector. We use the F-score to assess the performances of these two models in the aspect of salient sentence selection, and the number of selected controversial sentences to assess their performances in the controversial sentence reduction.

As shown in Table 5. We find that the model faces a significant performance drop in both the salient sentence selection and the controversial sentence reduction when the sentence opinion or believability is not considered. However, we find that BugSum-NOp significantly outperforms BugSum-NBeli on salient sentence selection. This is because in BugSum-NOp, the opinion of the evaluator sentence is not considered and all sentences are assumed to express positive opinions. Under this circumstance, sentences with higher number of evaluation behaviors in the bug report will be assigned with higher weights. In other words, BugSum-NOp assumes that sentences being discussed more often are more important. This

Table 5 Performance of control models

Dataset	Model	F-score	Selected controversial sentences
SDS	BugSum-NOp	0.425	14
	BugSum-NBeli	0.359	12
	BugSum	**0.495**	0
ADS	BugSum-NOp	0.412	4
	BugSum-NBeli	0.348	3
	BugSum	**0.492**	0
	BugSum-NOp	0.435	40
SDS	BugSum-NBeli	0.350	41
	BugSum	**0.495**	8

makes salient sentences with high frequency more likely to be selected, and results in a better performance. This is the domain reason why the factor believability can help improve BugSum's performance in salient sentence selection. In Bugsum, in addition to reducing the controversial sentences in the summary, the believability score assignment can help increase sentences that approved more often.

4.4.2 Influence of Informativeness

Informativeness refers to the amount of information contained in a generated summary. This factor is introduced to reduce the redundancy in the generated summary. As discussed in Section 2.2, bug reports contain various kind of information, such as bug reproduction and solution discussion. They account for different proportions of sentences in the bug report. Since previous works prefer information with a greater proportion, we assume that this may lead to redundancy of certain types of information in the summary and the absence of others. To validate our assumption, we study the informativeness of the summaries generated by previous works (PGS). As discussed in Section 3.5, we focus only on the bug reproduction and solution discussion, and count the proportion of sentences with these information types in the bug report (BR), the ground truth salient sentences, and PGS, respectively. Since the golden standard sentence set (GSS) we mentioned in Section 4.1.3 is consists of salient sentences selected manually, we take it as the ground truth of salient sentences in this study. Also, since the feature vectors used to calculate informativeness are weighted by believability, to avoid the interaction between these two factors affecting the results, we added the control model BugSum-NBeli, mentioned in Section 4.4.1, to the experiment. This study is deployed on bug reports from all three datasets (ADS, SDS, CDS), and the results are illustrated in Table 6. Note that the data in Table 6 represents not the number of sentences but the percentage. For example, the Bug Reproduction of BR in Table 6 is 8.9%. This means that the sentences related to Expected behavior in BR represent 8.9% of all sentences in BR.

We found that bug reproduction and solution discussion account for 8.9% and 43.1% of sentences in BR, and 11.2% and 48.3% of them in GSS, respectively. Although the proportion of sentences related to these two types of information slightly increased in GSS, their

Table 6 The proportion of sentences with different information types

	Bug reproduction	Solution discussion	
BR		8.9%	43.1%
GSS		11.2%	48.3%
PGS	Centroid	4.8%	73.2%
	DivRank	1.6%	63.4%
	Hurried	2.3%	62.9%
	BRC	3.7%	59.3%
	MMR	6.6%	57.7%
	Grasshopper	2.0%	61.0%
	ACS	3.9%	58.8%
	DeepSum	5.4%	63.4%
BugSum-NBeli		12.5%	53.5%
BugSum		**13.2%**	53.1%

proportions to each other remain similar to those in the bug report. However, in all PGS, sentences related to solution discussion accounted for at least 58.8% of the bug report, while sentences related to the bug reproduction account for less than 6.6%. The proportion of solution discussion in all PGS increased significantly compared to that in BR and GSS, but the proportion of bug reproduction in all PGS remain the same or even declined. Although sentences related to solution discussions have a greater proportion in the bug report, the over inclusion of such information may lead to the redundancy in the summary, and also the reduction of sentences with other information types due to the summary length limitation. This result validates our assumption in Section 2.2. Previous approaches usually measure the weight of a sentence based on the word frequency and prefer to select sentences with high weights. It results in sentences containing the majority information type are preferred to be selected into the summary. Under the length limitation, this further leads to redundancy of the majority information type in the summary as well as the absence of other types.

On the contrary, the information type proportions in the summary generated by BugSum are relatively more similar to those in GSS. It indicates that BugSum dose not over-select sentences of the majority information type, but reduces the proportion of such sentences and increased the number of sentences of other types. The similar performance of BugSum and BugSum-NBeli indicates that believability has almost no effect on informativeness. All these results indicate that, while ensuring the accuracy of the selected sentences, BugSum prefers to select sentences with different types of information due to its consideration of informativeness. Informativeness leverages the importance of the sentence and the redundancy it may introduce. It makes salient sentences with less majority type of information are more likely to be selected, and therefore improves the model performance.

Result 3: Both factors Believability and Informativeness play an important role in the performance of BugSum, and the lack of consideration of either factor will lead to a performance drop in the model. Believability not only reduces the impact of controversial sentences, but also increases that of sentences which are more approved. Meanwhile, Informativeness balances the number of sentences of different information types in the summary. Although Informativeness is counted using vectors weighted by Believability, Believability has nearly no Informativeness on Informativeness.

4.5 Answer to RQ4: Influence of Different SO-Model

As introduced in Section 3.2.2, BugSum uses the combination of DPCNN and random forest as SO-Model to assign the opinion scores of sentences. In order to have an assessment of the influence brought by such combination, we test the performance of several classification models and their combinations from three aspects, sentence opinion classification, controversial sentence reduction and salient sentence selection.

4.5.1 Selection and Combination of Classification Models

BugSum regards the sentence opinion assessment as a classification task, and uses the possibility of negative opinion to assign the opinion score of each sentence. In our experiments, we select two convolutional neural networks and three traditional classification algorithms as SO-Model. For convolutional neural networks, we include the TextCNN (Kim 2014) and DPCNN. TextCNN is widely used in text classification tasks. It uses multiple kernels of

different sizes to extract features from sentences. Meanwhile, DPCNN improves the structure of convolutional neural networks. It maintains a relatively low complexity and can effectively extract the long-distance relationship features in the text. As for traditional classification algorithms, we employ logistic regression (LR) (Kianifard and Kleinbaum 1995), random forest (RF), and support vector machine (SVM) (Vapnik 2000), which are also widely used algorithms in text classification.

4.5.2 Sentence Opinion Classification

As introduced in Section 3.2.2, SO-Model of BugSum is trained over a dataset consists of 3,000 sentences. These sentences are labeled with positive or negative to represent their opinions towards the evaluated sentences. We randomly select 80% of sentences from the dataset to train the models, and use the rest sentences as the test set. We repeat this process for 5 times, and use the average of the results as the final result.

As illustrated in Table 7, in our dataset, the performance of both the convolutional neural networks is higher than that of traditional classification algorithms. For the convolutional neural networks, DPCNN performs better than TextCNN. As for the traditional classification algorithms, LR has the best performance, while RF and SVM achieve competitive performance.

Most importantly, combining both types of algorithms can achieve better performance, compared with using only one classification algorithm. The combination of DPCNN and RF has the best performance. The performance of sentence opinion assessment increased from 81.1% to 91.2% compared with the SVM algorithm we used in previous work, and thus we use this combination as SO-Model in all our experiments. This result validates our assumption in Section 3.2.2. Convolutional neural networks compress features to be more concise and accurate, while traditional classification algorithms process input features more detail. Their combinations can incorporate both of their advantages so that they can achieve the best performance.

Since the input features of DPCNN are compressed through multiple layers, when combining DPCNN with RF, we also test the model's performance using different features that are compressed by different numbers of DPCNN layers. The number of layers of DPCNN

Table 7 Accuracy of sentence opinion classification

	SO-Model	Accuracy
CNN	TextCNN	0.889
	DPCNN	0.900
Traditional Algorithm	Random Forest (RF)	0.807
	Logistic Regression (LR)	0.773
	Support Vector Machine (SVM)	0.811
Combination	TextCNN + RF	0.901
	TextCNN + LR	0.894
	TextCNN + SVM	0.897
	DPCNN + RF	**0.912**
	DPCNN + LR	0.900
	DPCNN + SVM	0.910

depends on the length of the input data (Johnson and Tong 2017). As the length of the sentences after pre-processing in our datasets is around 30, we set the length of input data of DPCNN to 32, and accordingly set the number of layers to 5. As shown in Fig. 6, the performance of all classification models increases with the number of compressed layers. Features compressed by more layers in DPCNN can help achieve better performance when combining with traditional algorithms. Therefore, BugSum uses the compressed features from the last layer of DPCNN.

4.5.3 Controversial Sentence Reduction

Different SO-Model assigns different opinion scores to each sentence. It will further affect the assignment of sentence believability scores, according to Formula 1, and will then result in different numbers of selected controversial sentences, as shown in Table 8.

From Tables 7 and 8, we find that, in general, the more accurate the SO-Model is, the less controversial sentences are selected. Thus, the improvement of SO-Model can help the deduction of controversial sentences.

4.5.4 Salient Sentence Selection

We also test the performance of BugSum on the salient sentence selection.

As shown in Table 9, BugSum The combination of DPCNN and random forest achieves the best performance. But the changes of SO-Model have only a slight impact on the selection of salient sentences. The reason may be that,

The salient sentence selection in BugSum relies not only on the sentence believability, which is affected by SO-Model, but also on evaluation behaviors between sentences. We also find that SO-Model changes have a greater impact on datasets SDS and CDS (especially when using Logistic regression). This finding is consistent with our analysis in Section 4.2. SDS and CDS have a greater proportion of sentences covered by evaluation behaviors. As the sentence score is assigned by classification models, the change of classification models will have relatively greater impacts on these two datasets.

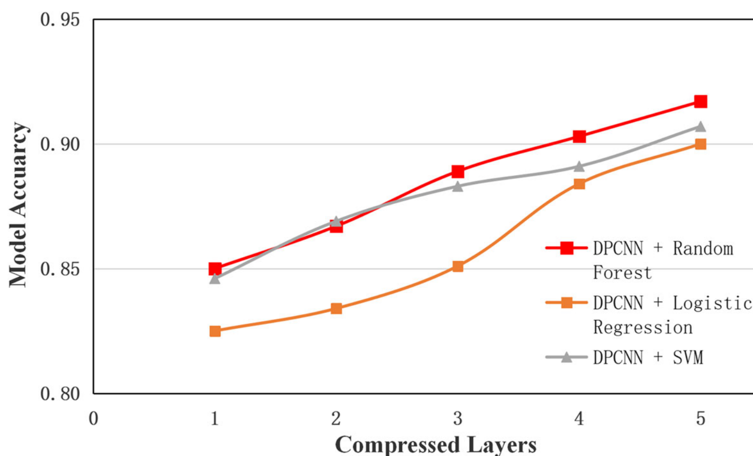

Fig. 6 The performance of models using different layers

Table 8 Reduction of controversial sentences using different so-model

	SO-Model	Selected controversial sentences
CNN	TextCNN	17
	DPCNN	14
Traditional Algorithm	Random Forest (RF)	21
	Logistic Regression (LR)	15
	Support Vector Machine (SVM)	21
Combination	TextCNN + RF	13
	TextCNN + LR	17
	TextCNN + SVM	14
	DPCNN + RF	**8**
	DPCNN + LR	13
	DPCNN + SVM	10

Result 4: The SO-Model has a great influence on the performance of BugSum. Combining convolutional neural networks with traditional classification algorithms, especially DPCNN with RF, can improve the permanence of BugSum, in terms of sentence opinion classification, salient sentence selection, and controversial sentence reduction.

4.6 Answer to RQ5: Influence of Parameters

BugSum contains three parameters: feature vector dimension, TF-IDF score 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.

Table 9 Performance of salient sentence selection using different so-model

	SO-Model	F-score		
		SDS	ADS	CDS
CNN	TextCNN	0.489	0.486	0.483
	DPCNN	0.492	0.488	0.494
Traditional Algorithm	Random Forest (RF)	0.471	0.482	0.462
	Logistic Regression (LR)	0.485	0.484	0.487
	Support Vector Machine (SVM)	0.473	0.479	0.460
Combination	TextCNN + RF	0.493	0.492	0.493
	TextCNN + LR	0.488	0.487	0.485
	TextCNN + SVM	0.490	0.491	0.493
	DPCNN + RF	**0.495**	**0.492**	**0.495**
	DPCNN + LR	0.492	0.489	0.494
	DPCNN + SVM	0.493	0.492	0.494

4.6.1 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. 7a, we present the *F-score* values of BugSum.

The performance curves of BugSum on SDS, ADS, and CDS exhibit a consistent 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 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 ADS, SDS, and CDS.

4.6.2 TF-IDF Score Threshold

As noted in Section 3.2.1, we identify the evaluation behaviors of ambiguous replies based on the sharing of topic-related words to assess the believability of sentences in the description.

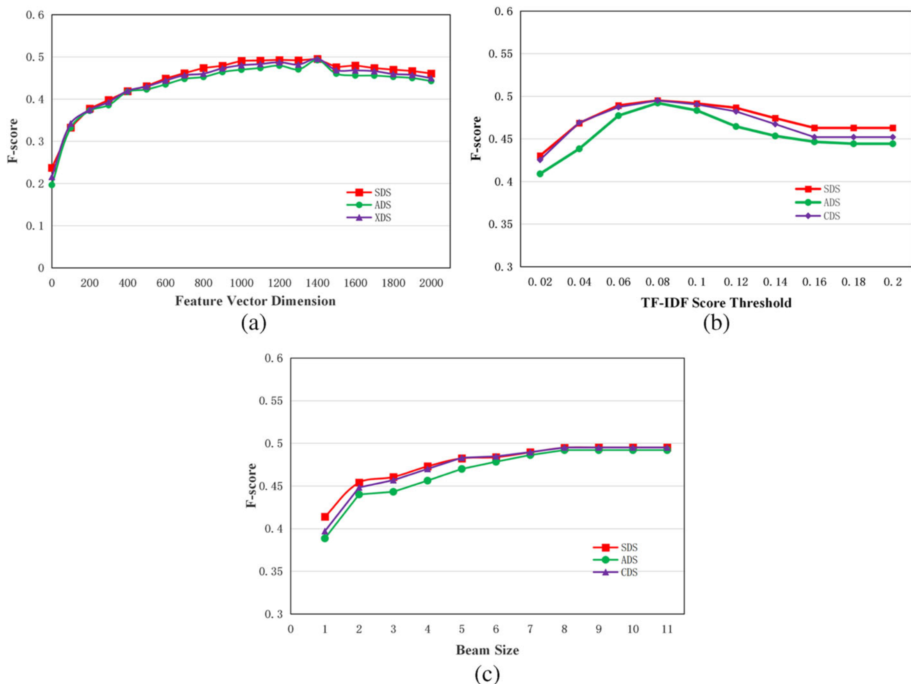

Fig. 7 Performance of BugSum influenced by different parameters

When selecting topic-related words, we need to set the threshold θ to the TF-IDF score. In this experiment, we test the sensibility of θ, from 0.02 to 0.20.

As can be seen in Fig. 7b, the performance of BugSum increases rapidly over all 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 topic-related 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 topic-related 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. 7b, the amount of topic-related words is too small, and the identified evaluation behaviors of ambiguous replies 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 and CDS, so the sentences in the description play a more critical role in ADS. Since the evaluation behaviors related to the description in our approach are mainly identified based on ambiguous replies, the noises introduced by θ will have more effects on ADS than on SDS and CDS. We also check the performance using other metrics and obtain similar results. Overall, we set the TF-IDF score threshold of topic-related word selection to 0.08 in all our experiments.

4.6.3 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. 7c, 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 5**: The dimension of the feature vector seriously affects the performance of BugSum. The threshold of TF-IDF score 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.

4.7 Answer to RQ6: Ablation Study

An ablation study usually refers to removing some components of the model, and seeing how that affects performance. In our approach, we implement the Sentence Feature Extraction (SFE) to extract semantic features from sentences, Dynamic Selection (DS) to select salient sentences, and Informativeness Enhancement (IE) to improve the informativeness of

the summary. In this section, we carry out an ablation study to test the effectiveness of these components against the commonly used alternative strategies.

Bag-of-Words (BoW) is one of the most popular representation strategies (Zhang et al. 2010), 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 (Li et al. 2018; Carbonell and Goldstein 1998; Radev et al. 2004; Lotufo et al. 2015; Mei et al. 2010; Zhu et al. 2007), 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 10. 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 semantic features in sentences include word frequency and word context. The BoW strategy can only preserve word frequency information, which leads to a performance drop. This also indicates that our approach can preserve domain semantic 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 informativeness of selected sentences. Alternative strategies like SSM tend to select sentences with redundant semantic features, and further cause relatively low *Recall* while achieving similar *Precision*.

For the informativeness enhancement, we compared the performance of BugSum on three datasets with and without this method. The results are shown in Table 11. BugSum-NIE refers to the performance of BugSum when informativeness enhancement (IE) is not used. Since the IE method makes the model select some extra sentences, we expand the number of selected sentences in BugSum correspondingly, and use Precision-Add to represent the accuracy of the additional selection. We find that the precision of additional selection (Precision-Add) is higher when using IE method, which indicate that IE method can improve the model's recall while maintaining accuracy. This verifies our assumption

Table 10 Performance of BugSum using different strategies

Dataset	Strategy		F-score	Precision	Recall	Pyramid	R-1	R-2
	BoW	SSM	0.297	0.410	0.246	0.303	0.441	0.092
SDS	BoW	DS	0.399	0.492	0.348	0.545	0.520	0.125
	SFE	SSM	0.386	0.519	0.311	0.542	0.514	0.116
	SFE	DS	**0.499**	**0.627**	**0.437**	**0.659**	**0.601**	**0.197**
ADS	BoW	SSM	0.292	0.400	0.249	0.490	0.457	0.112
	BoW	DS	0.386	0.454	0.356	0.563	0.517	0.210
	SFE	SSM	0.382	0.470	0.325	0.531	0.492	0.181
	SFE	DS	**0.494**	**0.611**	**0.424**	**0.691**	**0.568**	**0.272**
CDS	BoW	SSM	0.298	0.405	0.258	0.494	0.467	0.114
	BoW	DS	0.388	0.451	0.355	0.559	0.517	0.207
	SFE	SSM	0.380	0.466	0.326	0.525	0.492	0.182
	SFE	DS	**0.496**	**0.613**	**0.437**	**0.687**	**0.581**	**0.253**

Table 11 Performance of BugSum using informativeness enhancement

Dataset	Model	Precision	Recall	Precision-Add
SDS	BugSum-NIE	0.614	0.418	0.533
	BugSum	**0.627**	**0.437**	**0.622**
ADS	BugSum-NIE	0.607	0.420	0.492
	BugSum	**0.611**	**0.424**	**0.605**
CDS	BugSum-NIE	0.615	0.423	0.537
	BugSum	**0.617**	**0.427**	**0.614**

in Section 3.5: Due to the redundancy, over-selecting sentences of the major type makes it difficult to improve the informativeness of the summary, while selecting sentences with information lacking in the summary can achieve better results.

> **Result 6**: 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 all three datasets. The informativeness enhancement can improve the model's recall while maintaining fair accuracy.

5 Threats to Validity

In this section, we discuss threats to construct validity, internal validity, and external validity, which may affect the results of our study.

Construct Validity: The accuracy of selected controversial sentences is a threat to the validity of our experiments. In order to ensure the accuracy of selected controversial sentences, every selected sentence is confirmed by all 5 annotators. However, since none of the annotators have participated in the project of annotated bug reports, their understanding of these issues is limited. This makes it possible that there may be some controversial sentences in these bug reports that we do not find. This may affect the results of our experiment. However, since such controversial sentences are sporadic and can hardly affect the experiment results, such bias is acceptable in our experiments.

Internal Validity: The bias introduced by machine learning method is also a potential threat. On the sentence feature extraction and the assessment of sentence believability scores, we use the auto-encoder to extract textual features from the sentence, and the SO-Model to predict the sentence opinion. These models are based on machine learning, which is heavily relying on training data, and its performance can be significantly biased with different datasets. Moreover, even with the same training data, the performance of the machine learning method is not stable. It usually fluctuates from training to training. To mitigate these threats, we randomly divide the training and testing sets of the model. All experiments are run 5 times, and we use the average as the final result.

External Validity: The threat to external validity related to the generalizability of our findings and approach. To alleviate this threat, Our experiment study is conducted on the dataset

consists 34,140 bug reports from 8 popular open-source projects. Our approach has experimented against 8 baseline approaches on two public datasets and one customized dataset. The results suggest that our approach has acceptable generalizability.

6 Related Work

6.1 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. (2014) and Jiang et al. (2017) 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 (Lotufo et al. 2015), which requires the annotators to have certain expert knowledge and massive manual efforts. (Arya et al. 2019) labeled comments with their possible contained information, and let users choose corresponding sentences based on their requirements. Radev et al. (2004) 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 (Zhu et al. 2007; Mei et al. 2010) have attempted to select sentences according to reference relations, which were enhanced by a noise removal strategy designed by Mani et al. (2012). Lotufo et al. (2015) scored their sentences based on imitating human reading patterns, connected sentences according to their similarities, and chose sentences with the highest possibilities of being reached during a random traverse. Li et al. (2018) 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 supervised algorithm for bug report summarization that can efficiently reduce the possibility of controversial sentences been selected into the summary.

6.2 Summarization of NLP

Text summarization is one of the key applications of natural language processing for information condensation (Munot and Govilkar 2014). Wang and Cardie (2013) generated summaries for meeting records through templates, which required considerable manual effort to obtain. Cheng and Lapata (2016) transformed the bug summarization into a classification task, by using LSTM as a recurrent document encoder to represent documents. Nallapati et al. (2017) 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 and Rajan (2018) implemented the pointer network to add the salience of words into the prediction process. Narayan et al. (2018) optimized the *Rouge* evaluation metric through a reinforcement learning objective. Zhou et al. (2018) 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 (Nazar et al. 2016), *e.g.* source code and bug report.

6.3 Deep Learning in Software Engineering

In recent years, deep learning has been increasingly adopted to improve the performance of software engineering tasks (White et al. 2015). Moreno et al. (2013) and (Matskevich and Gordon 2018) utilized neural networks for source code analysis by integrating abstract

syntax trees (*i.e., AST*) and code semantic information to generate comments. Similarly, Wang et al. (2017) combined API sequence information with neural networks, and generated descriptions for object-related statement sequences. Moreover, Linares-Vásquez et al. (2015) and Buse and Weimer (2010) used neural networks to generate commit messages through extracting code changes. Jiang et al. (2017) improved the results of neural networks by adding filters to filter out the likely poor predictions. Liu et al. (2019) 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 (Fu and Menzies 2017; Hellendoorn and Devanbu 2017).

6.4 Text Classification

Text classification is a well-studied area in natural language processing, which is mainly based on traditional classification algorithms or deep learning approaches. Traditional classification algorithms such as Naive Bayes (Corporate AAFAI 1992), decision tree (Olanow and Koller 1998), and support vector machines (SVM) (Vapnik 2000) have simple structures and relatively few parameters, which makes them faster to be trained and easier to understand. (Surhone et al. 2010) improve the decision tree algorithm to mitigate the overfitting problem. Wang and Manning (2012) combine the Naive Bayes and SVM, and achieved better performance. Meanwhile, with the increasing recognition of deep learning, some approaches based on deep learning have been proposed to the text classification. (Kim 2014) applied the convolutional neural network (CNN) to text classification, which extracted domain features from sentences using kernels of different sizes. Johnson and Tong (2017) improved the structure of the naive CNN and proposed the Deep Pyramid Convolutional Neural Networks (DPCNN). It can extract the long-distance relationship features of sentences while maintaining relatively low complexity. Kim et al. (2018) incorporated word embeddings into the Open Directory Project (ODP) to handle large-scale text classification. These methods were based on and improved upon traditional classification algorithms or deep learning, and have achieved good performances in text classification. Recently, the combination of traditional classification algorithms with deep learning has become a new research trend. Wan et al. (2020) combined the decision tree with deep learning, which improved the model's performance and interpretability. Our approach combined random forest with DPCNN, and also achieved a significant improvement. The potential applications of the combination of traditional classification algorithms and deep learning remain to be explored.

7 Conclusion

In this study, we present a novel supervised summarization approach, that considers the semantic feature and sentence believability of bug report, and the informativeness of the summary. To generate more reliable and informative summaries for bug reports. To improve the performance of our previous design, we deeply explore the unique characteristics of bug reports and take ambiguous replies into consideration. For the purpose of validating the generality of our approach, we manually construct a controversial dataset, which contains 39 annotated bug reports and 62 controversial sentences. Extensive experiments over three public datasets show that the performance of our approach, compared to 8 typical baseline approaches, 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 points a promising direction for further work on conversation-based text analysis.

During our research, we find that, bug reports contain various types of information, such as bug reproduction, and possible solutions. Existing auto-generated summaries are concatenated by the salient sentences, while the complete information for each kind, such as complete steps to reproduce a bug, still needs to be organized by developers. In the future, we plan to extract a complete and believable summary for each kind of information in bug reports to provide more assistance to developers.

Acknowledgments This paper is supported by National Grand R&D Plan(Grant No. 2020AAA0103504), and National Natural Science Foundation (No.61872373 and No. 61672529).

References

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai

Arya D, Wang W, Guo JinLC, Cheng J (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, pp 454–464

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, pp 308–318

Bettenburg N, Premraj R, Zimmermann T, Kim S (2008) Extracting structural information from bug reports. In: Proceedings of the 2008 international working conference on Mining software repositories, pp 27–30

Bishnu PS, Bhattacherjee V (2012) Software fault prediction using quad tree-based k-means clustering algorithm

Bottou L (2010) Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010. Springer, pp 177–186

Buse RPL, Weimer WR (2010) Automatically documenting program changes. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, pp 33–42

Carbonell JG, Goldstein J (1998) The use of mmr, diversity-based reranking for reordering documents and producing summaries. In: SIGIR, vol 98, pp 335–336

Casalnuovo C, Vasilescu B, Devanbu P, Filkov V (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, pp 817–828

Cheng J, Lapata M (2016) Neural summarization by extracting sentences and words. arXiv:1603.07252

Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv:1406.1078

Christoffersen P, Jacobs K (2004) The importance of the loss function in option valuation. J Financ Econ 72(2):291–318

Corporate AAFAI (1992) Proceedings of the tenth national conference on artificial intelligence. In: Tenth National Conference on Artificial Intelligence

Fan Q, Yu Y, Yin G, Wang T, Wang H (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, pp 121–130

Fu W, Menzies T (2017) Easy over hard: A case study on deep learning. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 49–60

Hampe B (2002) Superlative verbs: A corpus-based study of semantic redundancy in english verb-particle constructions, vol 24. Gunter Narr Verlag

Han X, Yu T, Lo D (2020) Perflearner: Learning from bug reports to understand and generate performance test frames. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)

He H, Jia Z, Li S, Xu E, Liao X (2020) Cp-detector: using configuration-related performance properties to expose performance bugs. In: ASE '20: 35th IEEE/ACM International Conference on Automated Software Engineering

Hellendoorn VJ, Devanbu P (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, pp 763–773

Holm S (1979) A simple sequentially rejective multiple test procedure. Scand J Stat:65–70

Jadhav A, Rajan V (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), pp 142–151

Jiang H, Zhang J, Ma H, Nazar N, Ren Z (2017) Mining authorship characteristics in bug repositories. Sci China Inf Sci 60(1):012107

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 135–146

John, Anvik, Gail, C., Murphy (2011) Reducing the effort of bug report triage: Recommenders for development-oriented decisions. Acm Transactions on Software Engineering and Methodology

Johnson R, Tong Z (2017) Deep pyramid convolutional neural networks for text categorization. In: Meeting of the Association for Computational Linguistics

Kalliamvakou E, Damian D, Blincoe K, Singer L, German DM (2015) Open source-style collaborative development practices in commercial projects using github. In: Proceedings of the 37th International Conference on Software Engineering. IEEE Press, pp 574–585

Kianifard F, Kleinbaum DG (1995) Logistic regression: A self-learning text. Technometrics 37(1):116

Kim KM, Dinara A, Choi BJ, Lee SK (2018) Incorporating word embeddings into open directory project based large-scale classification. In: Pacific-asia Conference on Knowledge Discovery and13:15 2021/2/17 Data Mining

Kim W, Jeong O-R, Lee S-W (2010) On social web sites. Inf Syst 35(2):215–236

Kim Y (2014) Convolutional neural networks for sentence classification. Eprint Arxiv

Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105

LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324

Li X, Jiang H, Liu D, Ren Z, Li G (2018) Unsupervised deep bug report summarization. In: Proceedings of the 26th Conference on Program Comprehension. ACM, pp 144–155

Lin C-Y (2004) Rouge: A package for automatic evaluation of summaries. In: Text summarization branches out, pp 74–81

Lin H, Bilmes J (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, pp 912–920

Linares-Vásquez M, Cortés-Coy LF, Aponte J, Poshyvanyk D (2015) Changescribe: A tool for automatically generating commit messages. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 2. IEEE, pp 709–712

Liu Q, Liu Z, Zhu H, Fan H, Du B, Qian Y (2019) Generating commit messages from diffs using pointer-generator network. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, pp 299–309

Lotufo R, Malik Z, Czarnecki K (2015) Modelling the 'hurried'bug report reading process to summarize bug reports. Empir Softw Eng 20(2):516–548

Mani S, Catherine R, Sinha VS, Dubey A (2012) Ausum: approach for unsupervised bug report summarization. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p 11

Matskevich S, Gordon CS (2018) Generating comments from source code with ccgs. In: Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, pp 26–29

Mei Q, Guo J, Radev D (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, pp 1009–1018

Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: 2013 21st International Conference on Program Comprehension (ICPC). IEEE, pp 23–32

Munot N, Govilkar SS (2014) Comparative study of text summarization methods. Int J Comput Appl 102(12)

Nallapati R, Zhai F, Zhou B (2017) Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In: Thirty-First AAAI Conference on Artificial Intelligence

Narayan S, Cohen SB, Lapata M (2018) Ranking sentences for extractive summarization with reinforcement learning. arXiv:1802.08636

Nazar N, Hu Y, Jiang H (2016) Summarizing software artifacts: A literature review. J Comput Sci Technol 31(5):883–909
Nenkova A, Passonneau R, McKeown K (2007) The pyramid method: Incorporating human content selection variation in summarization evaluation. ACM Trans Speech Lang Process (TSLP) 4(2):4
Olanow CW, Koller WC (1998) An algorithm (decision tree) for the management of parkinson's disease: treatment guidelines. american academy of neurology. Neurology 50(3 Suppl 3):S1
Owczarzak K, Conroy JM, Dang HT, Nenkova A (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, pp 1–9
Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in pytorch
Porter MF (1980) An algorithm for suffix stripping. Program 14(3):130–137
Radev DR, Jing H, Styś M, Tam D (2004) Centroid-based summarization of multiple documents. Inf Process Manag 40(6):919–938
Ramos J 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, pp 133–142
Rastkar S, Murphy GC, Murray G (2010) Summarizing software artifacts: a case study of bug reports. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol 1. IEEE, pp 505–514
Rastkar S, Murphy GC, Murray G (2014) Automatic summarization of bug reports. IEEE Trans Softw Eng 40(4):366–380
Surhone LM, Tennoe MT, Henssonow SF, Breiman L (2010) Random forest. Mach Learn 45(1):5–32
Vapnik VN (2000) The nature of statistical learning theory. Springer
Wan A, Dunlap L, Ho D, Yin J, Lee S, Jin H, Petryk S, Bargal SA, Gonzalez JE (2020) Nbdt: Neural-backed decision trees. arXiv:2004.00221
Wang L, Cardie C (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, pp 1395–1405
Wang S, Manning CD (2012) Baselines and bigrams: Simple, good sentiment and topic classification. In: Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2
Wang X, Pollock L, Vijay-Shanker K (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, pp 205–216
White M, Vendome C, Linares-Vásquez M, Poshyvanyk D (2015) software repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp 334–345
Xuan J, Jiang H, Hu Y, Ren Z, Zou W, Luo Z, Wu X (2014) Towards effective bug triage with software data reduction techniques. IEEE Trans Knowl Data Eng 27(1):264–280
Zhang Y, Legunsen O, Li S, Dong W, He H, Xu T (2021) An evolutionary study of configuration design and implementation in cloud systems. In: the 43rd International Conference on Software Engineering
Zhang Y, Jin R, Zhou Z-H (2010) Understanding bag-of-words model: a statistical framework. Int J Mach Learn Cybern 1(1-4):43–52
Zhou Q, Yang N, Wei F, Huang S, Zhou M, Zhao T (2018) Neural document summarization by jointly learning to score and select sentences. arXiv:1807.02305
Zhu X, Goldberg A, Van Gael J, Andrzejewski D (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, pp 97–104
Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? IEEE Trans Softw Eng 36(5):618–643

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.