Unsupervised Deep Bug Report Summarization

Xiaochen Li
Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology, Dalian, China
li1989@mail.dlut.edu.cn

He Jiang
School of Software, Dalian University of Technology, Dalian, China; Beijing Institute of Technology
jianghe@dlut.edu.cn

Dong Liu
School of Software, Dalian University of Technology, Dalian, China
dongliu@mail.dlut.edu.cn

Zhilei Ren
School of Software, Dalian University of Technology, Dalian, China
zren@dlut.edu.cn

Ge Li
Key Laboratory of High Confidence Software Technologies (Peking University) Ministry of Education, China; Software Institute, Peking University, China
lige@pku.edu.cn

ABSTRACT

Bug report summarization is an effective way to reduce the considerable time in wading through numerous bug reports. Although some supervised and unsupervised algorithms have been proposed for this task, their performance is still limited, due to the particular characteristics of bug reports, including the evaluation behaviours in bug reports, the diverse sentences in software language and natural language, and the domain-specific predefined fields. In this study, we conduct the first exploration of the deep learning network on bug report summarization. Our approach, called DeepSum, is a novel stepped auto-encoder network with evaluation enhancement and predefined fields enhancement modules, which successfully integrates the bug report characteristics into a deep neural network. DeepSum is unsupervised. It significantly reduces the efforts on labeling huge training sets. Extensive experiments show that DeepSum outperforms the comparative algorithms by up to 13.2% and 9.2% in terms of F-score and Rouge-n metrics respectively over the public datasets, and achieves the state-of-the-art performance. Our work shows promising prospects for deep learning to summarize millions of bug reports.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**;

KEYWORDS

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

ACM Reference Format:
Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018. Unsupervised Deep Bug Report Summarization. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196321.3196326

1 INTRODUCTION

Bug repositories accumulate considerable knowledge for software projects [18, 47], including the experience on bug resolutions, historical software bugs, etc. To date, a single bug repository, e.g., the Eclipse Bugzilla repository, has already collected over 485,000 historical bug reports. Bug reports are important resources to maintain the long-term running of a software system, since the stakeholders of a software project prefer to understand the knowledge in these bug reports before conducting a software activity [33, 49]. For example, the common practice for software developers to fix newly reported bugs is to refer to similar historical bug reports for possible solutions [49]. Hence, nearly 600 sentences have to be read on average if a developer refers to only 10 historical bug reports [29]. Besides, bug reporters are usually required to wade through related bug reports before submitting a new one, to avoid a duplicate bug report submitted [33].

To reduce the tedious and time-consuming efforts in perusing historical bug reports, bug report summarization is proven to be a promising direction [38]. The Debian community even encourages stakeholders to manually set a summary for each bug report [7], though the considerable human costs may burden this activity. Hence, automatic extractive bug report summarization is an alternative way, which aims to extract salient sentences in a bug report. Previous studies try to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05. . . $15.00
https://doi.org/10.1145/3196321.3196326

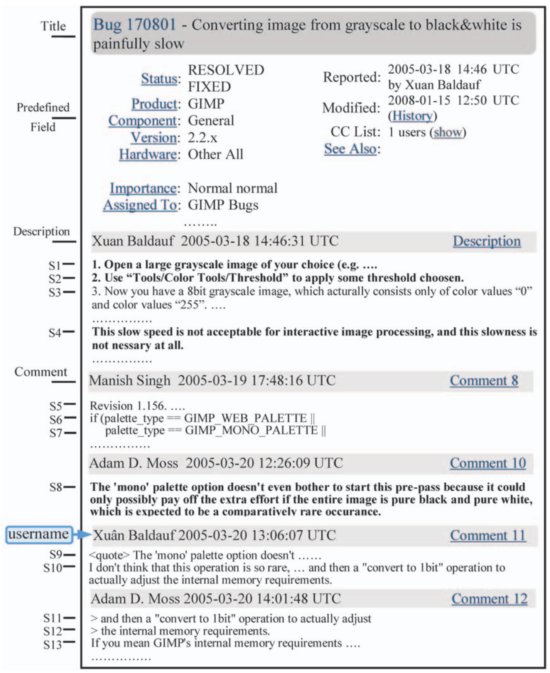

Figure 1: Example of Gnome bug report #170801.

train bug report summarizers with features from conversation-based text summarization [17, 38]. Due to the limitation of labeled data, unsupervised algorithms are also migrated for this task [27, 29]. However, their performance is still limited.

Compared with texts of news, biographies, etc., bug reports follow their own characteristics, which may weaken the effectiveness of a summarization technique. First, bug reports are conversation-based. Salient and duplicate sentences mix together due to the frequent evaluation or assessment behaviors [27]. Second, bug reports contain many different sentence types in natural language and software language [29]. It is non-trivial to automatically measure the contributions of different sentence types to the summary. Third, a bug report is usually associated with some predefined fields [52], e.g., the component or the product causing this bug, which may provide helpful information in identifying salient sentences.

Based on the above characteristics, we propose a novel *Deep* learning based *Sum*marizer, DeepSum, to better summarize bug reports. The kernel of DeepSum is a stepped auto-encoder network, which infers the bug report summary based on the hidden layers of the network. The basic idea of DeepSum is that, the hidden layers of a deep neural network is a compressed expression of the input feature vectors transformed from the sentences in the bug reports. The summary sentences of a bug report can be selected by measuring the weights of each sentence to this compressed expression with some sentence selection algorithms.

DeepSum first strengthens the vectors of the sentences being evaluated in the conversation-based reports, and filters the duplicate versions of the evaluated sentences with an evaluation enhancement module. Then, it stepwise feeds the vectors of different sentence types into an auto-encoder network to automatically measure the weights of words in distinct sentence types to the summary. The words in the predefined fields are enhanced when initializing the network parameters. At last, DeepSum summarizes the bug report based on the word weights by Dynamic Programming.

We compare DeepSum against seven previous summarization techniques over all the public datasets for bug report summarization [17, 27, 29, 38]. Extensive experiments show that DeepSum significantly outperforms the comparative algorithms by up to 13.2% and 9.2% in terms of F-score and Rouge-n metrics respectively over the public datasets. Most parameters of DeepSum can be set in a wide range of values. Meanwhile, the unsupervised nature of DeepSum makes it independent to the datasets and costs little time to label huge datasets for training.

In summary, we make the following contributions.

(1) To the best of our knowledge, it is the first attempt towards employing deep learning to summarize bug reports.

(2) We propose a novel deep neural network in DeepSum for bug report summarization.

(3) Experiments demonstrate that DeepSum achieves the state-of-the-art performance for bug report summarization.

2 MOTIVATION

Bug reports are widely used for recording software bugs. Typically, two roles are involved in writing a bug report, including a reporter (mostly a user or a tester) to submit and discuss the details of the bug, and the participators who are the developers interested in fixing the bug. Generally, a bug report consists of a title, a description, some predefined fields and several comments. Fig. 1 is a bug report example [2]. The title of this report concludes the report topic, namely the slow speed of converting images. The details for reproducing the bug are added in the description. The predefined fields show this bug happened in the "General" component of the "GIMP" product. To find satisfactory solutions, several comments are added to the report by the reporter and participators.

Facing numerous lengthy bug reports, bug report summarization aims to generate an summary by directly extracting and highlighting informative or salient sentences (also called summary sentences) from the description and comments of a bug report [38] (the bold sentences in Fig. 1). Although several supervised and unsupervised algorithms have been proposed to resolve the problem, the unique characteristics of bug reports may weaken the effectiveness of these techniques.

First, bug reports are conversation-based text with frequent evaluation behaviours [27]. The reporter and the participators discuss other's opinion by copying his/her sentence and adding evaluations to it. For example, in Fig. 1, before evaluating the sentence s_8, Xuân Baldauf wrote a similar sentence s_9 for evaluation. Another evaluation is that Adam D. Moss clicked the "reply" button to copy sentence s_{10} as s_{11} and s_{12} (sentences start with ">"), and then evaluated s_{10} in s_{13}. We call the sentences being evaluated as evaluated

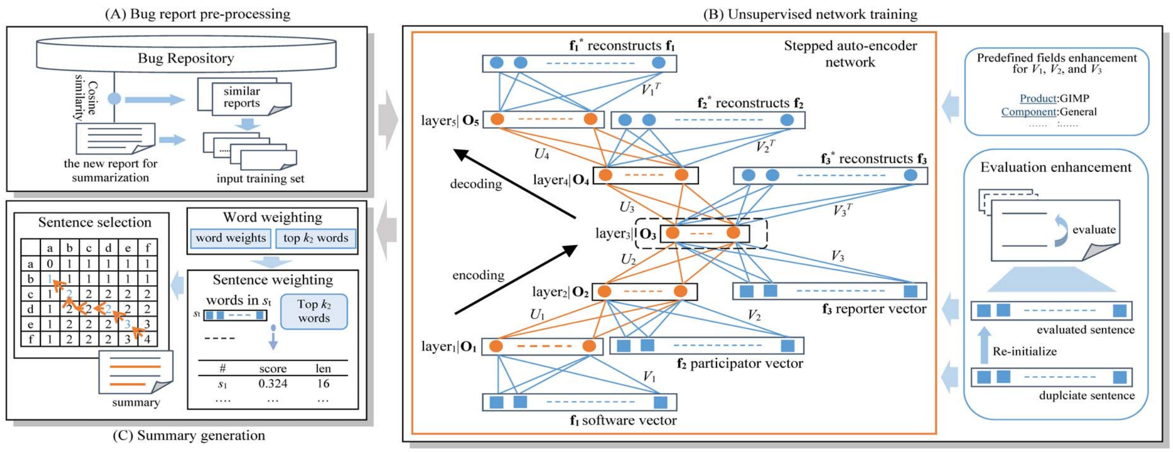

Figure 2: The framework of DeepSum.

sentences, e.g., s_8, s_{10}, and their similar versions as duplicate sentences, e.g., s_9, s_{11}, s_{12}. Although the evaluated sentences are frequently discussed and important, their corresponding duplicate sentences make it hard to discover the salient one.

Second, bug reports consist of different sentence types, namely the natural language sentences by the reporter, the natural language sentences by the participators, and the software language sentences (typically, code snippets and system messages). It is crucial to measure the contributions of different sentence types. Specifically, the natural language sentences by the reporter are usually more informative than that by the participators [27], since participators' comments are based on the topics proposed by the reporter. Meanwhile, despite developers are being familiar with the software language, a line in software language is usually less informative than that in natural language [29], e.g., a simple requirement "open a file" may result in many lines of code [42].

Third, a bug report is associated with many predefined fields [47, 52]. For these fields, *product*, *component*, *version*, and *hardware* are set by the reporter to reveal the environment information for reproducing the bug. Such information may be helpful for extracting salient sentences of a bug report.

Considering these unique characteristics, we propose DeepSum for effective bug reports summarization, a stepped auto-encoder network with evaluation enhancement and predefined fields enhancement modules to address these characteristics.

3 FRAMEWORK OF DEEPSUM

DeepSum summarizes a bug report by three steps, including bug report pre-processing, unsupervised network training, and summary generation. Given a new bug report, DeepSum first automatically detects a set of similar reports to form an unlabeled training set (Fig. 2(A)). The bug reports in the training set are used to train a stepped auto-encoder network for assigning sentence scores of the new bug reports (Fig. 2(B)). At last, DeepSum extracts summary sentences with Dynamic Programming based on these scores (Fig. 2(C)).

3.1 Bug Report Pre-processing

This step removes the noise in bug reports and collects similar reports for training the stepped auto-encoder network.

Bug reports are real-world data with considerable noises [49]. To remove the noises, DeepSum first tokenizes each sentence in the bug reports with a software-specific regular expression "[\w−]+(\.[\w−]+)*" [27]. Next, stop words removal [9] and Porter stemming [36] are conducted. At last, DeepSum filters sentences with less than three words [10], since they may not convey a piece of complete information.

After noise removal, DeepSum collects a set of bug reports for training the stepped auto-encoder network. For a new bug report (the bug report for summarizing), DeepSum first adds this new report to the training set. Since similar bug reports may contain common salient sentences or words to discuss bug solutions [4, 18], DeepSum also adds the top $k_1 - 1$ most similar reports prior to the new one from the same bug repository. The similarity is the cosine similarity between term frequency vectors of texts in description and comment fields of two bug reports calculated by Lucene [28]. We choose a simple and efficient way for calculation without consideration of the weight of each field or bug report topics.

At last, there are totally k_1 bug reports fed into the network. In this study, the size of the input training set k_1 is 100.

Example: Fig. 3(A) is an example representation of the new bug report in Fig. 1 and its top $k_1 - 1$ similar reports.

3.2 Unsupervised Network Training

This step trains a stepped auto-encoder network for summarization. We first transform the sentences in each training report into term frequency vectors and utilize evaluation enhancement to re-initialize the vectors. Then the vectors of different sentence types are fed into the network for training. In the network, words in predefined fields are enhanced during initializing the network parameters.

3.2.1 Evaluation Enhancement. Since there are frequent evaluation behaviours in bug reports, this sub-step enhances

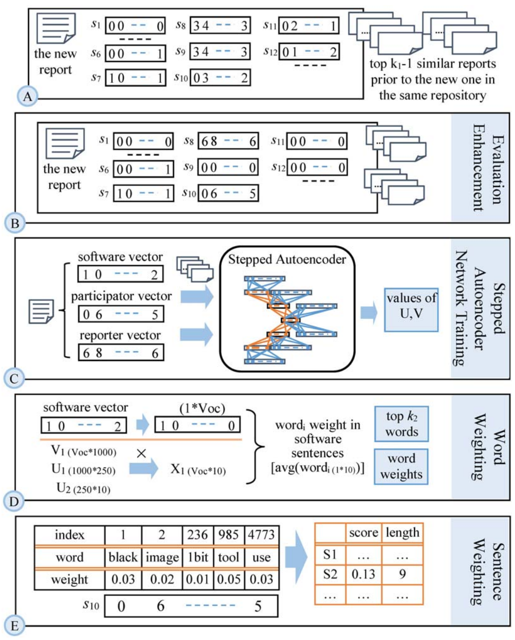

Figure 3: An example for DeepSum.

the evaluated sentences and reduces the influence of the duplicate versions for each training report. We represent a sentence s in a bug report as $\mathbf{f}_s = \left[f_1{}^s, f_2{}^s, \ldots, f_{voc}{}^s, \ldots, f_{|Voc|}{}^s\right]$, where $f_{voc}{}^s$ is the term frequency of the word f_{voc} in the sentence s normalized by the length of the bug report and Voc is the vocabulary of the training set. Under this representation, the similarity of two sentences is defined as the cosine similarity:

$$sim(\mathbf{f}_{s_i}, \mathbf{f}_{s_j}) = \frac{\mathbf{f}_{s_i} \cdot \mathbf{f}_{s_j}}{|\mathbf{f}_{s_i}|\,|\mathbf{f}_{s_j}|}, \tag{1}$$

where $\mathbf{f}_{s_i}$ and $\mathbf{f}_{s_j}$ are the term frequency vectors of the ith sentence s_i and the jth sentence s_j respectively.

Based on the above definition, the evaluated and duplicate sentences can be identified and re-initialized as follows.

(a) Identify a duplicate sentence. As shown in Fig. 1, a user usually clicks the reply button or writes similar sentences (e.g., the sentence s_9) to evaluate a previous sentence. When clicking the reply button, the Bugzilla system automatically copies and adds ">" to the sentences (e.g., the sentence s_{11}). DeepSum traverses sentences in a bug report from the last sentence to the first one. A duplicate sentence is detected, if the sentence starts with the sign ">" or its similarity with a previous sentence exceeds a threshold. The threshold θ is 0.9 as we will evaluate later in RQ1. These rules can be easily migrated after observing other bug repositories. Meanwhile, Bugzilla has been widely used by more than 136 companies and organizations to manage bug reports [24].

(b) Identify the related evaluated sentence. DeepSum identifies the previous sentence with the largest similarity of the duplicate one as the evaluated sentence. If two previous sentences have the same similarity, DeepSum identifies the sentence in front as the evaluated sentence.

(c) Re-initialize the two vectors. DeepSum adds the value of each element in the duplicate sentence vector to the evaluated sentence vector, and then sets all the elements in the duplicate sentence vector to be zero to reduce its influence.

The above process repeats until all the evaluated and duplicate sentences are identified. As a result, the elements in the evaluated sentence vector have larger initial values than those in the duplicate sentence vectors.

Example: Fig. 3(A) shows the initial sentence vectors of a new bug report. The vectors' length is the vocabulary of the training set. In Fig. 3(A), the first element in vector s_8 is three, which means the first word in the vocabulary occurs three times in s_8. For simplicity, we do not normalize these elements in the example. After evaluation enhancement, DeepSum identifies s_9 as a duplication of s_8, and regards s_{11} and s_{12} as duplications of s_{10}. Hence, the vector of s_9 is added to s_8, e.g., the first element in vector s_8 is re-initialized to 6 (in Fig. 3(B)). Similarly, s_{10} is re-set with s_{11} and s_{12}. Then, the elements in vectors s_9, s_{11} and s_{12} are set as zero.

3.2.2 Stepped Auto-encoder Network Training. With the enhanced vectors of each training bug report, DeepSum trains a stepped auto-encoder network for summarization.

As analyzed in Section 2, bug reports usually consist of three types of sentences which may have distinct importance to the summary, including the software language sentences (referred as software sentences), the natural language sentences by participators (participator sentences), and the natural language sentences by the reporter (reporter sentences). DeepSum detects the sentence types as follows.

(a) Detect the software sentences. Software sentences are detected with the bug report analysis framework Infozilla [3]. For code snippets, it first identifies code lines by matching some strong regular expressions, e.g., "=.*?;$", and then expands the regions with surrounding sentences by many weak rules, e.g., sentences contain "class", "public", etc. The system messages are identified as the continuous sentences containing ":" or "_". Besides, if the distance of two regions is within three sentences, we also take the sentences between them as software language, in order to include any exceptions to the above rules. We find that 95% detected software sentences are true positive in our experiment.

(b) Detect the reporter sentences and participator sentences. DeepSum directly classifies the remaining natural language sentences by matching the reporter's name with the *username* item in the description and comment fields.

After detecting the sentence types, DeepSum stepwise encodes and decodes different sentence types with a stepped auto-encoder network. In the following paragraphs, we introduce the inputs and outputs, the architecture, and the training process of the network.

(a) The network inputs and outputs. The network inputs are three vectors. For a training bug report, DeepSum adds up the sentence vectors of the same sentence types to form three input vectors. These vectors denote as the software vector $\mathbf{f}_1$, the participator vector $\mathbf{f}_2$, and the reporter vector $\mathbf{f}_3$. After encoding and decoding the inputs with the network, DeepSum requires the outputs of the network to be three vectors of the same length with the inputs, namely $\mathbf{f}_1{}^*$,$\mathbf{f}_2{}^*$, $\mathbf{f}_3{}^*$, which means the reconstruction of the input vectors. The objective of the network is to minimize the differences between the input vectors and the output vectors.

(b) The network architecture. As shown in Fig. 2(B), there are five hidden layers in the network. The unit number of each hidden layer is similar with a traditional autoencoder network [26], namely num_{layer_1}=num_{layer_5}=1000, num_{layer_2}=num_{layer_4}=250, and num_{layer_3}=10. For this architecture, we have the following observations:

First, the network compresses three input vectors into 10 hidden units in layer_3, and then reconstructs the entire input vectors to the outputs. Hence, this hidden layer can be viewed as a compressed expression of the input vectors, which meets the concept of summarization. The compressed expression may reserve the meaningful information of the inputs to infer informative words of a new bug report.

Second, the network stepwise feeds the three inputs into the hidden layers. The software vector is at the bottom and the reporter vector is near the hidden layer layer_3. The reason is that, since software sentences are usually less informative than the other sentence types, DeepSum heavily compresses the software vector with three hidden layers, i.e., layer_1 to layer_3, to filter the noises. In contrast, the reporter sentences are usually more informative, DeepSum only compresses the reporter vector once to reserve all meaningful information.

(c) The training process. DeepSum inputs each training bug report to the network. By minimizing the differences between the inputs and outputs, DeepSum optimizes the network parameters with the widely used RMSProp optimizer [13]. The initial learning rate η of the optimizer is 0.01 [19]. We also apply the dropout strategy [40] to prevent the network from overfitting. The dropout rate is 0.5 [40].

In this sub-step, DeepSum first encodes the inputs with layers $layer_1$ to $layer_3$. Let $\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$ be the software vector, participator vector, and reporter vector respectively, $\mathbf{O}_i$ be the output vector of $layer_i$ (i≤3), and $\mathbf{b}_i$ be the bias of $layer_i$. The matrices U_i and V_i are the weighs of connections between the fully-connected $layer_{i-1}$ or $\mathbf{f}_{i-1}$ and $layer_i$. We have

$$\mathbf{O}_i = sig\left(\mathbf{O}_{i-1}U_{i-1} + \mathbf{f}_i V_i + \mathbf{b}_i\right), \tag{2}$$

where $1 \leq i \leq 3$, $\mathbf{O}_0 = \mathbf{0}$, $U_0 = \mathbf{0}$, and $sig(\cdot)$ is the sigmoid activation function. Then, DeepSum decodes the hidden layer $layer_3$ with $layer_4$ and $layer_5$. Let $\mathbf{f}_1{}^*$, $\mathbf{f}_2{}^*$, $\mathbf{f}_3{}^*$ be the outputs corresponding to $\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$, $\mathbf{O}_i$ $(4 \leq i \leq 5)$ represents the reconstruction of hidden layer $layer_1$ and $layer_2$, and $\mathbf{b}_i$ and $\mathbf{c}_j$ are the bias of $\mathbf{O}_i$ and $\mathbf{f}_j{}^*$ respectively. We have

$$\mathbf{O}_i = sig\left(\mathbf{O}_{i-1}U_{i-1} + \mathbf{b}_i\right), \tag{3}$$

$$\mathbf{f}_j{}^* = sig\left(\mathbf{O}_{6-j}{V_j}^T + \mathbf{c}_j\right), \tag{4}$$

where $U_{i-1} = {U_{6-i}}^T, 1 \leq j \leq 3$.

DeepSum optimizes the network parameters by minimizing the cross-entropy loss function [46] between $\mathbf{f}$=[$\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$] and $\mathbf{f}^*$=[$\mathbf{f}_1{}^*$, $\mathbf{f}_2{}^*$, $\mathbf{f}_3{}^*$]. The parameters continue optimizing until the minimum loss retains unchanged within 100 iterations.

Example: In Fig. 3(C), DeepSum merges the vectors of the same sentence type. For the vectors in Fig. 2(B), it merges s_6 and s_7 into the software vector, merges s_8, s_{11}, and s_{12} into the participator vector, and merges s_1, s_9, and s_{10} into the reporter vector. Then, DeepSum feeds the three vectors of each training report into the network to train U and V.

3.2.3 Predefined Fields Enhancement.

Some predefined fields set by the reporter may be helpful for revealing a bug, including *product*, *component*, *version*, and *hardware*. DeepSum strengthens the influence of words in these fields in the same way as [26] when initializing the network parameters. Initially, the matrixes U, V are randomly initialized to a zero mean Gaussian with a standard deviation of 0.01 [26]. If the ith word in the input vectors exists in the predefined fields, the parameters in row i of V are set to the maximum value in V. The reason is that, DeepSum assigns word weights according to the values of U and V (discussed in Section 3.3). The weights of the words in the predefined fields may be enhanced after maximizing the initial values of U and V.

3.3 Summary Generation

This step assigns weights for words in the new bug report and collects a set of salient words. The salient words are used to assign scores for the bug report sentences. With the sentence scores, a summary is generated by Dynamic Programming.

3.3.1 Word Weighting.

DeepSum assigns word weights according to the input vectors of the new bug report and the parameters U and V. The intuitive idea is that, we assume the hidden layer ($layer_3$ in this study) is a compressed expression or summary of the input vectors. Hence, we can assign word weights by measuring how each word in the input vectors contributes to this compressed expression.

Specifically, we explain this process by an example of assigning the weight of word i in the software vector (in Fig. 3(D)). We first collect the trained network parameters U and V, and the input vectors of the new bug report. Then we multiply V_1, U_1 and U_2 to generate a new matrix X_1. For meaningful multiplication, a sigmoid function is applied on the matrices to transform the matrix values to be positive. The row number of X_1 is the same as the software vector length, and the column number is the unit number of $layer_3$, i.e., 10. X_1 is regarded as a simplified function to transform the software vector to $layer_3$. In this study, we utilize U and V to calculate the transformation function, since they are the core parameters of the network.

Then we assign the weight of word i. We set all the values in the software vector to be zero except word i. In Fig. 3(D), the first element "1" in the vector is reserved. We can multiply this vector with X_1 to get the importance of word i to each of the ten hidden units in $layer_3$. We average the ten values to achieve the weight of word i in software sentences. Similarly, the weights of word i in participator sentences and reporter sentences can be calculated by V_2 and U_2, or V_3. DeepSum sums the three weights of word i, and ranks and selects the top k_2 words in the new bug report as a salient words set.

3.3.2 Sentence Weighting. Based on the the salient words set, we assign scores for the sentences in the new bug report. For each sentence, we collect its enhanced sentence vector, and multiple the elements in the vector with the summed weights of the corresponding word in the salient words set.

Example: Fig. 3(E) lists the five salient words (k_2=5), their indexes, and the summed word weights in the example. Words "black" and "image" are the first two element in the vocabulary (indexed as 1 and 2), and the word "use" is the last one (indexed as 4773). In the example, the sentence score of s_{10} is achieved by a multiplication between the enhanced term frequencies in the sentence vector of s_{10} (in Fig. 3(B)) and the summed word weights in the salient words set, namely s_{10} is scored as 0*0.03+6*0.02+...+5*0.03. All the weights of words excluded in the salient words set are zero.

3.3.3 Sentence Selection. With the sentence scores, bug report summarization turns into selecting sentences $\boldsymbol{s}_{\text{select}}$ in the new bug report to maximize the total sentence score under a length limitation. It is a typical 0-1 Knapsack problem, which can be solved by Dynamic Programming [43], a common sentence selection algorithm in text summarization [26, 31]. Similar to previous studies [27, 38], we select sentences until they reach 25% length of the new bug report in words. The idea of Dynamic Programming is to decide whether to add s_i to the summary under the remaining summary length limitation, when the maximum total sentence score of s_0 to s_{i-1} is already achieved.

4 EXPERIMENT SETUP

DeepSum is implemented with the machine intelligence library Tensorflow [1]. It runs on Ubuntu 16.04 with Intel Core(TM) i7-6700 CPU, GTX1080 GPU and 16G memory.

4.1 Dataset

We evaluate the summarizers over two public bug report datasets, namely SDS (Summary DataSet) [38] and ADS (Authorship DataSet) [17], consisting of 36 and 96 bug reports respectively. Each bug report in the datasets is annotated by three annotators. The annotators are asked to conclude the report in around 25% length of the report in their own words. The concluded sentences are called an *Abs*tractive *Ref*erence summary *AbsRef*. Then the annotators link each sentence in *AbsRef* to one or more sentences in the original bug report. The *Ext*ractive *Ref*erence summary *ExtRef* consists of the sentences linked by at least two annotators.

4.2 Baseline Algorithms

We compare DeepSum with seven algorithms in previous studies. These algorithms summarize bug reports from multiple aspects, including taking advantage of the conversation-based characteristic of bug reports (BRC[38], ACS[17]), transferring classical text summarizers to bug reports (Centroid, MMR, Grasshopper, DivRank) [29], and manually mining features from bug reports for summarization (Hurried [27]).

We reproduce the baseline algorithms, since previous studies evaluate their algorithms under different criteria [29, 38]. In the experiments, an algorithm selects summary sentences until they reach 25% length of the new bug report in words [17, 27, 38]. For the supervised algorithms BRC and ACS, we conduct the Leave-One-Out (LOO) framework [16, 38] and Two-Fold Cross-Validation (TFCV) framework [41] for evaluation, denoted as BRC_{LOO} and BRC_{TFCV}, and ACS_{LOO} and ACS_{TFCV}. LOO framework [38] trains a summarizer by all the reports in the same dataset except the one for summarization. TFCV framework [41] randomly splits the dataset into two parts for training and testing. LOO framework requires more labeled training data. Due to the lack of public labeled datasets for bug report summarization, TFCV framework may better reflect the actual performance of the supervised algorithms.

4.3 Evaluation Metrics

We first evaluate the algorithms with four metrics proposed in previous studies for bug report summarization [38], namely *Precison*, *Recall*, *F-score*, and *Pyramid score*. For a set of selected summary sentences $\boldsymbol{s}_{\text{select}}$, these metrics are:

$$Precision = Num_{\text{hit}}/Num_{\text{selected}}, \tag{5}$$

$$Recall = Num_{\text{hit}}/Num_{\text{ExfRef}}, \tag{6}$$

$$F\text{-}score = \frac{2 * Precision * Recall}{Precision + Recall}, \tag{7}$$

$$Pyramid = Num_{\text{TotalLinks}}/Num_{\text{MaxLinks}}, \tag{8}$$

where Num_{selected} is the number of sentences in $\boldsymbol{s}_{\text{select}}$, Num_{hit} is the number of sentences in $\boldsymbol{s}_{\text{select}}$ which belongs to *ExtRef*, Num_{ExtRef} is the number of sentences in *ExtRef*, $Num_{\text{TotalLinks}}$ is the total number of times that sentences in $\boldsymbol{s}_{\text{select}}$ are linked by the annotators, while Num_{MaxLinks} is the maximum possible links for the same number of sentences.

We further evaluate the algorithms with the widely accepted summarization evaluation package Rouge-1.5.5 [23], which evaluates $\boldsymbol{s}_{\text{select}}$ with the human written summaries *AbsRef*. In this study, the metrics[1] R-1 and R-2 are used, due to their abilities in emulating human evaluation procedures [5, 34].

$$Rouge\text{-}n = \frac{\sum_{s \in \text{AbsRef}} \sum_{gram_n \in s} Count_{\text{match}}(gram_n)}{\sum_{s \in \text{AbsRef}} \sum_{gram_n \in s} Count(gram_n)}, \tag{9}$$

where n is the n-gram length, $Count_{\text{match}}(gram_n)$ is the number of n-gram co-occurring in both $\boldsymbol{s}_{\text{select}}$ and *AbsRef*. For metrics R-1 and R-2, the values of n are 1 and 2 respectively.

4.4 Research Questions (RQ)

RQ1: Do the parameters influence DeepSum's performance?
RQ2: How do the evaluation enhancement and predefined field enhancement modules influence DeepSum's performance?
RQ3: Is DeepSum effective in assigning word weights compared to some alternative strategies?
RQ4: How does DeepSum perform against baselines?
RQ5: How does DeepSum perform under different sentence selection criteria?

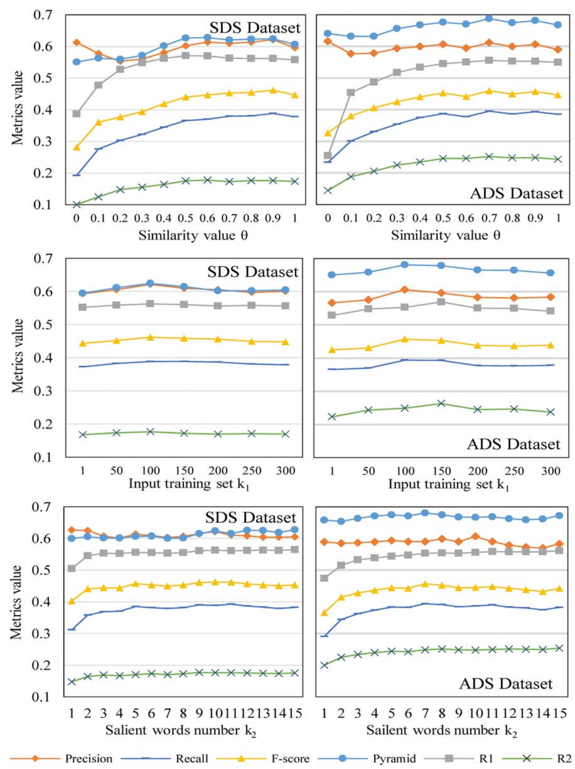

Figure 4: Influence of different parameters.

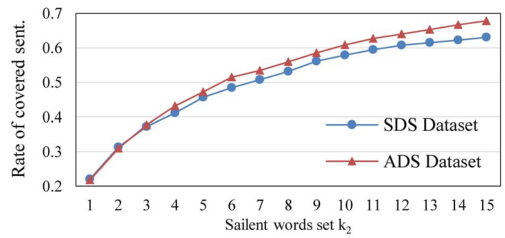

Figure 5: Rate of covered sentences over distinct k_2

5 EXPERIMENT RESULTS

5.1 Answer to RQ1: Parameter Influence

Fig. 4 shows the influence of similarity value θ, training set size k_1, and salient words number k_2 with respect to distinct evaluation metrics over the two datasets.

5.1.1 Similarity Value θ. DeepSum detects the duplicate versions of an evaluated sentence with θ. To tune θ, we fix k_1=100 and k_2=10, and vary θ from 0 to 1 with a step size of 0.1. When θ=1, no similar sentences are considered as duplications. When θ=0, all the sentences are regarded as the duplication of the first sentence. Since the first sentence tends to be important in a bug report, when θ is small (θ <0.3), *Precision* is high. However, other metrics are low as all the other sentences are regarded as the duplications with zero element vectors. When θ increases from 0.6 to 0.9, the metrics turn to be stable. We set θ=0.9 in the following experiments.

5.1.2 Training Set Size k_1. DeepSum expands the new bug report with similar ones in the same repository. To tune k_1, we set θ=0.9 and k_2=10. When k_1=1, it means only the new bug report for summarization is fed into the deep learning network which may overfit the network. When k_1 increases to 300, the performance of DeepSum begins to drop slightly, since many unrelated bug reports are included in the training set. DeepSum performs best when k_1 ranges from 100 to 150 over both datasets. The reason may be that, these reports are similar with the new bug reports in both words distribution and salient words or sentences, which may be helpful to measure the salient words for the new bug report. We set k_1=100 in the experiments according to the observation.

5.1.3 Salient Words Number k_2. The salient words number is used to assign sentence scores in the new bug report. To investigate k_2, we set θ=0.9 and k_1=100, and k_2 varies from 1 to 15 with a step size of 1. When k_2=1, it means DeepSum weights the sentences only based on the word of the largest weight in the bug report. Although this word may repeatedly occur in the *ExtRef* summary, it loses the diversity of the summary, which leads to a relatively high *Precision* but low *Recall*. As k_2 increases, the performance is relatively stable, namely DeepSum is insensitive to k_2. According to this observation, we set k_2=10 in the experiments.

For k_2, we avoid to set a large value. First, we find as k_2 increases, long sentences with large amount of words tend to have high sentence scores, which makes DeepSum fail to select the right summary sentences. Second, when k_2=10, DeepSum can cover the majority of sentences in a bug report. We calculate the rate of covered sentences over distinct k_2 in Fig. 5. We count the number of sentences that sentence scores are greater than zero and divided this number by the number of sentences in a dataset. When k_2=1, DeepSum can assign scores to around 20% sentences in the datasets, namely 80% sentence scores are zero. Thus, it is hard to rank and select the 80% sentences. When k_2=10, around 60% sentences are assigned by DeepSum. Since the remaining sentences may be filtered during pre-processing or not contain any informative words, DeepSum is able to select the correct summary sentences then.

Conclusion. By setting these parameters, DeepSum can handle diverse situations as different metrics are preferred, e.g., *Precision* or *Recall*. Most parameters of DeepSum can be set in a wide range of values.

5.2 Answer to RQ2: Module Influence

DeepSum utilizes a stepped auto-encoder network with evaluation enhancement module and predefined field enhancement module to take advantages of bug report characteristics. Table 1 shows the influence of these modules. The sign "A" denotes the evaluation enhancement module and "B" denotes the predefined fields enhancement module. The sign "×" means running DeepSum without a certain module.

[1]Rouge options [5]: -c 95 -r 1000 -n 2 -m -u -x -f A -p 0.5 -t 0

Table 1: Influence of Different Modules.

	A	B	Precision	Recall	F-score	Pyramid	R1	R2
SDS	√	√	**0.621**	**0.388**	**0.462**	**0.624**	**0.563**	**0.177**
	×	√	0.533	0.336	0.397	0.551	0.543	0.160
	√	×	0.600	0.381	0.450	0.598	0.552	0.166
	×	×	0.529	0.339	0.399	0.544	0.540	0.160
ADS	√	√	**0.606**	**0.394**	**0.457**	**0.681**	**0.553**	**0.249**
	×	√	0.586	0.382	0.443	0.661	0.548	0.243
	√	×	0.577	0.371	0.432	0.661	0.548	0.246
	×	×	0.558	0.359	0.418	0.639	0.543	0.242

Table 2: Performance on Word Weighting Strategies.

		Precision	Recall	F-score	Pyramid	R1	R2
SDS	TF Strategy	0.583	0.380	0.445	0.581	0.556	0.167
	AE Strategy	0.581	0.370	0.437	0.590	0.555	0.165
	DeepSum	**0.621**	**0.388**	**0.462**	**0.624**	**0.563**	**0.177**
ADS	TF Strategy	0.573	0.360	0.425	0.581	0.544	0.240
	AE Strategy	0.577	0.366	0.427	0.657	0.549	0.240
	DeepSum	**0.606**	**0.394**	**0.457**	**0.681**	**0.553**	**0.249**

Obviously each module has its own contribution to the final summary. The evaluation enhancement module identifies frequently discussed sentences in a bug report and filters the duplicate copies. The predefined fields enhancement module leverages the information in the predefined fields to identify the salient words. In addition, when we remove both modules, the performance is the worst which means these modules are both useful for summarizing bug reports.

We find that in SDS, removing the evaluation enhancement module leads to a sharp drop on most metrics, e.g., F-score drops from 0.462 to 0.397. In contrast, this module has less impact on ADS. The reason is that, the average number of sentences per bug report in SDS is 65. The evaluation behaviours happen frequently in these long bug reports. While, the average number of sentences is only 39 in ADS. Most bugs are fixed before the evaluation behaviours happen. However, the results show that the modules of DeepSum work well regardless of datasets with distinct bug reports length.

Conclusion. The evaluation enhancement module and predefined fields enhancement module have positive influence on DeepSum. These modules work well over different datasets.

5.3 Answer to RQ3: Word Weighting

This RQ investigates whether the word weighting strategy of DeepSum in Section 3.3.1 is useful. For this purpose, we replace the word weighting strategy with two alternative ones and keep the other modules of DeepSum.

TF Strategy. The first one is a Term Frequency based strategy. It investigates whether a deep neural network is necessary. For a new bug report, TF strategy first transforms the sentences in the report into vectors in the same way as DeepSum. For fair comparison, the evaluation enhancement module is applied on each sentence vector. Then we calculate and rank the words according to their term frequency. The top 10 words are selected as the salient words. The weights of words are the values of term frequency. Based on the selected words and enhanced sentence vectors, TF strategy assigns sentence scores in the same way as DeepSum.

AE Strategy. The second one is an Auto-Encoder based strategy. It investigates whether the stepped auto-encoder network outperforms a standard auto-encoder network in word weighting. Compared with DeepSum in Fig. 2(B), a standard auto-encoder only has one input layer $\mathbf{f}_1$ and output layer $\mathbf{f}_1{}^*$, i.e., we remove the inputs $\mathbf{f}_2$, $\mathbf{f}_3$ and the outputs $\mathbf{f}_2{}^*$, $\mathbf{f}_3{}^*$. AE strategy works as follows. After collecting the input training set, we merge all the sentence vectors of different sentence types into a single vector and feed it to $\mathbf{f}_1$. AE strategy conducts the same training and sentence selection procedure as DeepSum. The evaluation enhancement and predefined field enhancement modules are also applied. Unlike DeepSum, AE strategy assigns weights of all words based on a three layer compression, namely assigning word weights based on V_1, U_1, U_2 and the single input vector.

Table 2 shows the performance of different word weighting strategies. DeepSum outperforms both alternative strategies. For example, DeepSum has a relative improvement by up to 5.7% and 7.5% in terms of F-score over the two datasets. The results show that the stepped auto-encoder network is superior to a simple word weighting strategy in assigning word weights, i.e., TF strategy. Meanwhile, words in different sentence types make distinct contributions to the summary. Compared with a standard auto-encoder network (AE strategy), DeepSum fully leverages the sentence type information to make a more accurate measurement on words.

Conclusion. DeepSum's word weighting strategy outperforms the alternatives, i.e., TF strategy and AE strategy.

5.4 Answer to RQ4: Overall Performance

We compare DeepSum with previous algorithms for bug report summarization [17, 27, 29, 38]. Table 3 and Table 4 show the overall performance of DeepSum averaged by ten times' running. A bold value is the best result among all the algorithms. A grey cell means DeepSum outperforms an algorithm with $p<0.05$ by the paired Wilcoxon signed rank test under Holm's correction [14].

Overall, DeepSum outperforms the comparative algorithms in terms of the majority of the metrics. It achieves the state-of-the-art results on 4 and 6 metrics over the two datasets. On the SDS dataset, DeepSum outperforms the other algorithms by up to 0.119 and 0.092 in terms of F-score and R-1 respectively. On the ADS dataset, DeepSum still achieves equal or better performance than the comparative algorithms. Although different metrics evaluate an algorithm from their unique aspects, DeepSum outperforms these algorithms in most cases and is stable on distinct metrics and datasets.

We note that the training sets heavily affect the supervised algorithms. The ACS algorithm performs well on ADS in a Leave-One-Out framework, since ADS contains numerous labeled reports written by the same reporter which fits for ACS. However, neither such a training set or the training framework is always available. Hence, when it comes to S-DS that contains few reports by the same reporter, ACS

Table 3: Overall Comparison on SDS.

	Precision	Recall	F-score	Pyramid	R-1	R-2
BRC_{LOO}	0.570	0.350	0.400	0.630	0.521	0.140
BRC_{TFCV}	0.524	0.321	0.362	0.580	0.493	0.130
ACS_{LOO}	0.595	0.337	0.400	0.604	0.516	0.135
ACS_{TFCV}	0.562	0.310	0.359	0.572	0.488	0.126
Centroid	0.536	0.269	0.343	0.460	0.471	0.126
MMR	0.617	0.353	0.429	0.551	0.498	0.145
Grasshopper	0.525	0.300	0.368	0.521	0.505	0.135
DivRank	0.591	0.301	0.378	0.546	0.500	0.139
Hurried	**0.710**	0.300	0.410	**0.710**	0.525	0.153
DeepSsum	0.621	**0.388**	**0.462**	0.624	**0.563**	**0.177**

Table 4: Overall Comparison on ADS.

	Precision	Recall	F-score	Pyramid	R-1	R-2
BRC_{LOO}	0.568	0.350	0.412	0.659	0.517	0.201
BRC_{TFCV}	0.528	0.314	0.388	0.620	0.492	0.180
ACS_{LOO}	0.605	0.391	0.452	0.671	0.546	0.235
ACS_{TFCV}	0.556	0.343	0.400	0.625	0.520	0.211
Centroid	0.488	0.280	0.337	0.561	0.473	0.183
MMR	0.505	0.356	0.395	0.585	0.503	0.206
Grasshopper	0.446	0.337	0.362	0.548	0.504	0.201
DivRank	0.445	0.282	0.325	0.545	0.498	0.202
Hurried	0.580	0.349	0.418	0.637	0.544	0.241
DeepSsum	**0.606**	**0.394**	**0.457**	**0.681**	**0.553**	**0.249**

drops significantly. In addition, if ACS runs in a Two-Fold Cross-Validation framework, it still performs poorly. This phenomenon can be also observed from BRC_{LOO} and BRC_{TFCV}. In contrast, DeepSum is robust to the labeled training sets.

Another observation is that all the results on R-2 are low. The reason is that R-2 metric measures the 2-gram overlap between the selected sentences and the human written *AbsRef* summaries. Since the *AbsRef* summary is written by annotators according to their understandings of a bug report, there is no 100% match between the *AbsRef* summary and the selected sentences. However, R-2 is still important in evaluation [5]. In this metric, DeepSum outperforms most comparative algorithms in a statistic sense.

At last, we find that, Hurried achieves high Precision on SDS, but loses its dominance to DeepSum on ADS. The reason may be that, bug reports in ADS are relatively short. There are not much sentiment information and evaluation information which are required by Hurried. Hence, DeepSum has an over 9% relative improvement on F-score to Hurried.

Conclusion. DeepSum shows promising performance for summarizing bug reports over distinct evaluation metrics.

5.5 Answer to RQ5: Summary Length

Fig. 6 presents the performance of DeepSum on varied summary lengths. The x-axis is the summary length and y-axis shows the values of different metrics. We generate a summary from 15% to 70% length of the bug report [27].

As shown in Fig. 6, DeepSum works well under different summary lengths. As the summary length increases, Precision of DeepSum begins to drop, since only a small ratio of sentences in a bug report belongs to the *ExtRef* summary.

Table 5: Top Words for Bug Report #170801.

Sentence type	Five salient words
$\mathbf{f}_1$: software sent.	palette colormap gimpimageconvert.c palette_typ option
$\mathbf{f}_2$: participator sent.	palette image convert use difference
$\mathbf{f}_3$: reporter sent.	black operation image 1bit tool
Final	black image 1bit tool use

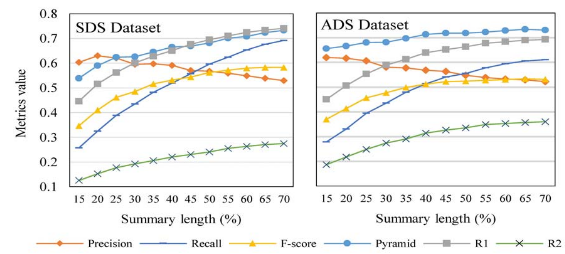

Figure 6: Results on varied summary lengths.

However, the variation of Precision is small (<10%). In contrast, there is a more than 40% promotion on Recall. The Recall values reach 0.692 and 0.611 on SDS and ADS respectively when we select a summary of 70% bug report length. It implies that DeepSum successfully ranks the sentences in the *ExtRef* summary higher than the other sentences. Similarly, other metrics also increase. For example, F-score significantly increases to 0.532 for SDS and 0.512 for ADS at 40% length. Then, F-score is stable with less than 5% increasement when the summary length further increases.

However, we note that there is no an optimal summary length in practise, due to the different reading habits of developers. Hence, after assigning the scores of sentences in a bug report, DeepSum is able to efficiently generate the summary of different lengths by Dynamic Programming.

Conclusion. DeepSum works well at different summary lengths. With a summary of 40% length, DeepSum can cover more than half of the sentences in the ground-truth summary.

6 DISCUSSION

6.1 Example by DeepSum

Taking the bug report in Fig. 1 as an example, Table 5 lists the top 5 words weighted by DeepSum with respect to different sentence types. The network summarizes bug reports in three aspects. For the input of software sentences, the top 5 words are variables (e.g., colormap, palette_type) or file names (e.g., gimpimage-convert.c) regarding the code snippets. The important words of the participator sentences focus on the solutions (e.g., use, different, palette) to the reported bug. While the salient words of the reporter sentences address the core problems of the report, namely conversing *1bit image* between *black* and white is slow. DeepSum stepwise weights sentences of different types according to their contributions to the summary. As a result, most information of the reporter's sentences is reserved (e.g., black, 1bit); some information of the participators' solutions is supplemented (e.g., use);

Table 6: A Partial Summary by DeepSum.

Title	(Gnome Bug Report #170801) Converting image from grayscale to black&white is painfully slow
#	Sentences
1	1. open a large grayscale image of your choice (e.g. larger than 2000*2000 pixels, maybe a scan result from your scanner)
2	2. use "tools/color tools/threshold" to apply some threshold choosen arbitrarily.
3	image where to 1bit conversion is either slow or buggy (with gimp 2.2.4)
4	i'm attaching a test case image. this may be a related bug.
5	the 'mono' palette option doesn't even bother to start this pre-pass because it could only possibly pay off the extra effort if the entire image is pure black and pure white, which is expected to be a comparatively rare occurance.
6	"threshold" operation, and then a "convert to 1bit" operation to actually adjust the internal memory requirements.

and massive information of the software language is filtered. Since every sentence type may contribute to the summary, DeepSum automatically decides the filtered information in this process. It shows DeepSum's ability in measuring words according to different sentence types of bug reports.

Table 6 is a partial summary for the bug report in Fig. 1 by DeepSum. In practice, these sentences are labeled in bold on the original bug report for developers to understand their context. The summary is in accordance with the above observations. Sentences #1-#3 are the phenomena and reproducing steps of the bug, while sentences #4-#6 are the possible solutions and some important discussions between the reporter and participators.

6.2 Threats to Validity

The generality of DeepSum should be further studied. To alleviate this threat, we evaluate DeepSum over all the public datasets for bug report summarization. We also compare DeepSum with seven previous summary algorithms over multiple metrics.

The running time of DeepSum is another threat in real developing scenarios [15]. DeepSum is a deep neural network based algorithm which takes 5.66 minutes on average to summarize a bug report, including 5.57 minutes in training the network. Since bug report summarization is usually used for perusing historical bug reports, most reports could be summarized in an off-line mode. Meanwhile, bug reports in several years ago may receive less attention due to the changes of the software architecture and source code. Additionally, computing in the cloud could also shorten the running time.

7 RELATED WORK

7.1 Bug Report Summarization

Bug report summarization techniques can be classified into supervised and unsupervised ones. For supervised techniques, Rastkar et al. migrate features from email summarization to train bug report summarizers [38]. They find that labeling domain-specific data sets is important to improve the performance of bug report summarizers. Jiang et al.[17] summarize bug reports in consideration of the reporters' authorship.

Due to the limitation of labeled data, many unsupervised text summarization and sentence selection strategies are also employed [29], including Centroid [37], Maximal Marginal Relevance (MMR) [6], Grasshopper [51], and Diverse Rank (DivRank) [30]. Lotufo et al. [27] propose a graph-based unsupervised algorithm by simulating human reading behaviors, which achieves relatively high Precision by sacrificing Recall.

In this study, DeepSum is a novel unsupervised algorithm for bug report summarization, which assigns weights of words and sentences without human mined features.

7.2 Deep Neural Networks for Software

Recently, several studies utilize deep neural networks to represent software artifacts. Mou et al. [32] build a tree-based convolutional neural network to conduct programming representation. Peng et al. [35] propose several "coding criteria" for better leveraging neural networks for software artifacts.

Based on the novel program representation, Wang et al. [44] and Li et al. [22] utilize convolutional neural networks to predict the bug-prone source code. Similar ideas are also employed by Yang et al. [50] for just-in-time defect prediction. After defect prediction, Lam et al. [20, 21] conduct bug localization with deep neural networks and Gupta et al. [12] automatically fix software bugs. In addition, White et al. [45] conduct code completion with deep neural networks. Gu et al. [11] recommend API usage sequences with an attention based recurrent neural network. Raychev et al. [39] and Chen et al. [25] train deep neural networks to synthesize codes and If-Then programs respectively.

Besides encoding source code, a few studies encode software texts in natural languages. Deshmukh et al. [8] detect duplicate bug reports with neural networks. Xu et al. [48] analyze Stack Overflow posts by convolutional neural networks.

In contrast to previous studies, we attempt to apply a novel deep neural networks to summarize bug reports, a complex software artifact with natural languages and source code.

8 CONCLUSION

Bug reports are crucial to fix software bugs. In this study, we propose an unsupervised deep learning algorithm for bug report summarization. Our model fully leverages the characteristics of bug reports. Experiments over two public bug report datasets show that our model outperforms the comparative algorithms significantly by adopting domain-specific characteristics. In the future, we plan to conduct case studies to investigate whether such an automated model could facilitate millions of software developers in perusing bug reports in a real developing scenario. More information about DeepSum is at http://oscar-lab.org/people/~xcli/open/deepsum/

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under Grants 61722202, 61772107.

REFERENCES

[1] Tensorflow an open-source software library for Machine Intelligence. 2017. https://www.tensorflow.org/. (2017).

[2] Xuân Baldauf. 2005. Converting image from grayscale to black&white is painfully slow. https://bugzilla.gnome.org/show_bug.cgi?id=170801. (2005).

[3] Nicolas Bettenburg, Rahul Premraj, Sunghun Kim, and Thomas Zimmermann. 2008. Extracting structural information from bug reports. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR'08)*. ACM, 27–30.

[4] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful really?. In *IEEE International Conference on Software Maintenance (ICSM'08)*. IEEE, 337–345.

[5] Ziqiang Cao, Furu Wei, Li Dong, Sujian Li, and Ming Zhou. 2015. Ranking with recursive neural networks and its application to multi-document summarization.. In *AAAI Conference on Artificial Intelligence (AAAI'12)*. 2153–2159.

[6] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 335–336.

[7] Debian. 2016. Introduction to the bug control and manipulation mailserver. http://www.debian.org/Bugs/server-control#summary. (2016).

[8] Jayati Deshmukh, Annervaz K M, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques.. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*.

[9] Damian Doyle. 2017. Default English stopwords list. http://www.ranks.nl/stopwords. (2017).

[10] Laura V Galvis Carreño and Kristina Winbladh. 2013. Analysis of user comments: an approach for software requirements evolution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. IEEE Press, 582–591.

[11] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, 631–642.

[12] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI Conference on Artificial Intelligence (AAAI'17)*. 1345–1351.

[13] Geoffrey Hinton and Tijmen Tieleman. 2012. Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning. (2012).

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

[15] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? automatic cause analysis for test alarms in system and integration testing. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 712–723.

[16] He Jiang, Jingxuan Zhang, Xiaochen Li, Zhilei Ren, and David Lo. 2016. A more accurate model for finding tutorial segments explaining APIs. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 157–167.

[17] He Jiang, Jingxuan Zhang, Hhongjing Ma, Nazar Najam, and Zhilei Ren. 2017. Mining authorship characteristics in bug repositories. *Science China Informaction Science* 58 (2017).

[18] Sunghun Kim, Kai Pan, and EE Whitehead Jr. 2006. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. ACM, 35–45.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS'12)*.

[20] AnNgoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17)*. IEEE Press, 218–229.

[21] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 476–481.

[22] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software Defect Prediction via Convolutional Neural Network. In *IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 318–328.

[23] Chin-Yew Lin. 2004. Rouge: a package for automatic evaluation of summaries. In *Text summarization branches out: Proceedings of the ACL-04 workshop*, Vol. 8. Barcelona, Spain.

[24] Bugzilla Installation List. 2017. https://www.bugzilla.org/installation-list/. (2017).

[25] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems (NIPS'16)*. 4574–4582.

[26] Yan Liu, Sheng-hua Zhong, and Wenjie Li. 2012. Query-oriented multi-document summarization via unsupervised deep learning.. In *AAAI Conference on Artificial Intelligence (AAAI'12)*.

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

[28] Apache Lucene. 2016. http://lucene.apache.org/. (2016).

[29] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM, 11.

[30] Qiaozhu Mei, Jian Guo, and Dragomir Radev. 2010. Divrank: the interplay of prestige and diversity in information networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'10)*. ACM, 1009–1018.

[31] Hajime Morita, Ryohei Sasano, Hiroya Takamura, and Manabu Okumura. 2013. Subtree extractive summarization via submodular maximization.. In *Annual Meeting of the Association for Computational Linguistics (ACL'13)*. Citeseer, 1023–1032.

[32] Lili Mou, Ge Li, lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293.

[33] Mozilla. 2013. Bug writing guidelines. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_writing_guidelines. (2013).

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

[35] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 547–553.

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

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

[38] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering (TSE'14)* 40, 4 (2014), 366–380.

[39] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

[40] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[41] Yuan Tian, David Lo, and Chengnian Sun. 2013. Drone: Predicting priority of reported bugs by multi-factor analysis. In *IEEE International Conference on Software Maintenance (ICSM'13)*. IEEE, 200–209.

[42] Tjekkles. 2011. Java: Open a file (Windows + Mac). https://stackoverflow.com/questions/7024031/. (2011).

[43] Paolo Toth. 1980. Dynamic programming algorithms for the zero-one knapsack problem. *Computing* 25, 1 (1980), 29–45.

[44] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 297–308.

[45] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR'15)*. IEEE, 334–345.

[46] Fen Xia, Tie Yan Liu, Jue Wang, Hang Li, and Hang Li. 2008. Listwise approach to learning to rank: theory and algorithm. In *International Conference on Machine Learning*. 1192–1199.

[47] Xin Xia, David Lo, Emad Shihab, and Xinyu Wang. 2016. Automated bug report field reassignment and refinement prediction. *IEEE Transactions on Reliability* 65, 3 (2016), 1094–1113.

[48] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. ACM, 51–62.

[49] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering (TKDE'15)* 27, 1 (2015), 264–280.

[50] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *IEEE International Conference on Software Quality, Reliability and Security (QRS'15)*. IEEE, 17–26.

[51] Xiaojin Zhu, Andrew B Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving diversity in ranking using absorbing random walks.. In *Proceedings of NAACL HLT*. 97–104.

[52] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering (TSE'10)* 36, 5 (2010), 618–643.