IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 7, 2019, accepted November 30, 2019, date of publication December 3, 2019, date of current version December 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2957424

Deep Code-Comment Understanding and Assessment

DEZE WANG[1], **YONG GUO**[2], **WEI DONG**[1], **ZHIMING WANG**[1], **HAORAN LIU**[1], **AND SHANSHAN LI**[1]

[1]College of Computer Science, National University of Defense Technology, Changsha 410073, China
[2]College of Systems Engineering, National University of Defense Technology, Changsha 410073, China

Corresponding author: Yong Guo (yguo@nudt.edu.cn)

This work was supported by NSFC under Grant 61690203, Grant 61872373, and Grant 61872375.

ABSTRACT Code comments are a key software component for program comprehension and software maintainability. High-quality code and comments are urgently needed by data-driven models widely used in tasks like code summarization. Many existing approaches for assessing the quality of comments are machine learning based classification algorithms or rely on heuristic rules. These approaches are difficult to capture the complicated features of text data and are often limited in accuracy, efficiency, and generalization ability. In this paper, we convert the quality assessment of code comments into a classification problem based on the multi-input neural network. We summarize the input, the code and comments, into vectors using the attention-based Bi-LSTM model and the weighted GloVe model, respectively, and concatenate the code vectors and the comment vectors as the input of the Multiple-Layer Perceptron classifier for the comment quality assessment. Experimental results show that our approach, in general, outperforms the previous technique, on both our labeled dataset and the public dataset, with the F1-score of 96.91% and 91.90%, respectively. Using the training set and the testing set from distinct sources, our approach can still achieve reasonable performance, which demonstrates its generalization ability.

INDEX TERMS Code comment, source code, multi-input neural network, text classification.

I. INTRODUCTION

Code comments are essential to software systems and can help developers better understand the code for modification or reuse. Previous studies have shown that on average developers spend around 60% of their time on program comprehension activities [1] and the commented code is easier to understand than the uncommented code [2]. Developers and users can therefore save a lot of time with code comments. Code comments also play an important role in software maintenance. They provide key information to understand systems to be maintained [3] and the lack of them can significantly reduce software maintainability [4].

In recent years, data-driven models in various tasks like code summarization, code completion, and comment generation have been analyzed and studied by many researchers. These models require high-quality code and comments, to learn knowledge from them. For example, Wong *et al.* [32] generate code comments automatically by analyzing existing software repositories. They use the comments from some code segments to describe the other similar code segments. The quality of the existing comments is important to their method. Unfortunately, comments may be irrelevant or inconsistent with the source code they refer to. To this end, an effective quality assessment approach of comments is of great importance, and with this approach developers can pick out high-quality comments and their corresponding code to train and test data-driven models.

Many assessment approaches of comment quality have been proposed. Some traditional methods distinguish the quality of comments by defining various metrics [6], [7], such as comment consistency, maintainability, readability, etc. However, the cost of the manual work is always large, thus for modern systems with growing scale of code comments, these approaches may not be applicable. Machine learning classifiers have also been used to assess the quality of code comments [5], [9], [10], [33]. These approaches cannot achieve high enough accuracy because of the complexity of the code and comments and the insufficiency of the model expression ability.

Unlike the general text, comments and code have unique characteristics that make sentence classification for general

The associate editor coordinating the review of this manuscript and approving it for publication was Xinyu Du.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

texts not applicable. First, comments are closely related to the code. The correlation between the code and comments should be considered in the quality assessment of comments. However, the code and comments are the programming language and the natural language, respectively. The different structural features and rules of these two languages make it difficult to capture the correlation of the code and comments. Second, the words and identifiers introduced in the code and comments require the domain knowledge to understand. Finally, the language forms of the code and comments vary from software projects due to the using of various programming styles and preferences. It is crucial to extract the key information from raw data.

The basic idea in this paper is that we treat the quality assessment of comments as a classification problem. We use a classifier to decide if the quality of a snippet of code comment is reliable. Due to limited public datasets [31], we select the Java projects uploaded to GitHub before October 2018 and arrange for three annotators to spend three months in labeling the method-level code and comments for further model training and testing. We perform domain-specific data cleaning techniques to build our dataset from the labeled raw data. For the code and comments with different data characteristics, we propose *DComment*, a Multi-Input Neural Network to summarize them into vectors based on their characteristics. We vectorize tokens in the code and comments using the enhanced GloVe model, which is pretrained with the domain-specific corpus. For the code, we train a Bi-LSTM model to summarize the token vectors and introduce the attention mechanism to focus on the tokens with the important information. For comments, we sum up the token vectors as the comment representations, with the consideration of token weights. The token weights are calculated based on TF-IDF. Finally, we feed the code vectors and the comment vectors into the Multiple-Layer Perceptron (MLP) classifier.

We compare *DComment* with the state-of-the-art previous work on our labeled dataset and the public dataset [9]. Experiments show that, in general, *DComment* outperforms previous approaches over all metrics including precision, recall and F1-score. *DComment* achieves the F1-score of 96.91% on our labeled dataset and 91.90% on the public dataset. To evaluate the generalization ability of *DComment*, we conduct experiments on *DComment* with the training data and the testing data from different sources, and the F1-score of 85.80% is achieved. The evaluation results demonstrate that *DComment* performs well on the new unseen data and has high generalization ability.

In this paper, we make the first attempt to apply deep neural networks to the automatic quality assessment of code comments. Our main contributions are as follows:

- We manually label 2156 method-level code-comment pairs from Java projects uploaded to GitHub before October 2018 and establish a public dataset.[1]
- We propose a framework for the automatic quality assessment of code comments, based on the extraction of the distinct characteristics of the code and comments.
- We evaluate and compare the performance of *DComment* against other approaches, using our labeled dataset and the public dataset. Experimental results show that *DComment* achieves the state-of-the-art performance in comment classification and has high generalization ability.

The rest of the paper is organized as follows. Section II presents the related work, Section III introduces our data preparation process, and Section IV describes our framework *DComment* and its implementation details, followed by Section V that presents the evaluation results. Finally, Section VI concludes the paper.

II. RELATED WORK

A. COMMENT CLASSIFICATION

Previous work can be classified as traditional methods and machine learning methods. For traditional methods, Khamis *et al.* [8] proposed the JavadocMiner tool for analyzing the quality of Javadoc comments and they used a set of heuristic rules to determine the quality of comments according to both the comment content and the consistency between the code and comments. Tan *et al.* [11] also targeted Javadoc, parsing Javadoc comments to infer the properties of the corresponding code. A random test case was then generated to identify the inconsistencies between the code and comments. Based on the hypothesis ''if the code is high quality, then the comments give a good description of the code'', Lawrie *et al.* [12] used the information retrieval technique to evaluate the code quality by the cosine similarity on the vector space model. Padioleau *et al.* [6] designed a classification system based on the meaning of the comments. This semantic analysis is a great breakthrough compared with previous work, but manual classification is time-consuming and cannot be widely applied. Wen *et al.* [7] presented an empirical study about the co-evolution of the code and comments, and defined a taxonomy that categorized the types of code-comment inconsistencies fixed by developers.

For machine learning methods, Steidl *et al.* [5] applied machine learning methods to evaluate the comments quality in terms of effectiveness, consistency, completeness, and usability, and to classify comments into seven high-level categories. Pascarella and Bacchelli [13] further refined the Java code comment classification, and designed a Bayesian classifier to predict the comment categories. Corazza *et al.* [9] manually evaluated the consistency of 3636 methods of three open source softwares, and proposed an SVM classifier on the vector space model to automatically classify comments. Liu *et al.* [10] proposed a machine learning-based approach that utilized 64 different features to detect comments that should be updated when the code changes. Shinyama *et al.* [30] proposed eleven distinct categories of code comments and developed a decision-tree based classifier

[1] It is available at https://github.com/wangdeze18/Code-Comment-Assessment-Dataset.

that could identify the explanatory comments. Yu *et al.* [33] propose a code comment quality assessment approach by using the aggregation of the basic classification algorithms. They evaluate comments in terms of their formats, language forms, contents and relevance with the code. Their methods do a preliminary study on automatic comment classification, but the accuracy of these methods can be further improved. They mainly focus on comment forms, and lack the deep understanding of code structure and comment semantics during the assessment process.

B. DEEP LEARNING FOR SOFTWARE ENGINEERING

Many studies combined software engineering with deep learning, and applied deep learning to different software artifacts. Deep learning has achieved competitive performance against previous algorithms on many software engineering tasks like code summarization, code completion, and comment generation.

Researchers conducted many studies based on code representation. Mou *et al.* [14] used an RNN Encoder-Decoder model to generate code from natural language user intentions. White *et al.* [15] applied the RNN language model to source code files and demonstrated their effectiveness at a real software engineering task, code suggestion. Gu *et al.* [16] proposed a deep neural network, which jointly embedded code snippets and natural language descriptions into vector space for code search. In addition to embedding source code, there are some studies focusing on natural language. Deshmukh *et al.* [17] used neural networks to detect duplicate bug reports. Yin *et al.* [18] proposed a method for extracting aligned code/natural language pairs from the Q&A website, Stack Overflow. Fucci *et al.* [19] studied how well modern text classification approaches can automatically identify the documentation that contains specific knowledge types. Deep learning achieves encouraging success in the field of software engineering for its powerful ability to feature extraction and learning. In this paper, we try to combine data processing techniques with end-to-end deep learning methods on the code and comment artifacts, and design a framework for automatic quality assessment of comments.

III. DATA PREPARATION

Due to improper programming habits of developers and code evolution, there are always many problems within the comments, such as outdated, inconsistent with their corresponding code, etc. Moreover, few datasets of method-level code-comment pairs are currently available. Therefore, we manually annotate a set of code-comment pairs for training and evaluation. Fig. 1 shows the overview of our data preparation process. Next, we introduce the details of the data preparation process and our labeled dataset.

A. DATA FILTERING

The projects we select were uploaded to GitHub before October 2018 with the tag of "Java". To ensure the reliability of the code and comments, we only include projects with

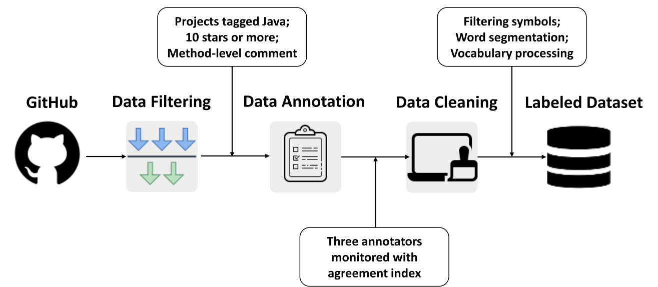


FIGURE 1. An overview of the data preparation process.

10 stars or more. In this paper, we focus on method-level comments, since they are likely to have corresponding code snippets. We exclude too short comments with less than 4 tokens, because they may have very limited information and are not worthy for the learning process of our model.

B. DATA ANNOTATION

Three annotators spend three months annotating the method-level code and comments of the projects collected from GitHub to build a labeled dataset. Each comment and its corresponding code can form a code-comment pair. For each pair, one annotator independently labels the comment with the value of non-coherent or coherent, according to the content of the comment and the correlation of the code and the comment.

For the same code-comment pair, different annotators may assign different labels, due to their subjectivity. To solve this problem, we assign at least two annotators to assess each code-comment pair.

If the judgments of the annotators differ, we introduce the kappa index [20] to estimate the agreement. It is a statistical measure of the consistency of the annotators on a classified project. The definition of κ is:

$$\kappa = \frac{P_0 - P_e}{1 - P_e} \tag{1}$$

where P_0 is the relative observed agreement among annotators, and P_e is the hypothetical probability of chance agreement.

Values greater than 0.75 are considered as a good agreement [21]. If values are lower than 0.75, annotators discuss the results and find out the reasons for major disagreements. After discussion and arguments, a common label was given. Finally, all the comments are given a label.

C. DATA CLEANING

In order to ensure the accuracy of the quality assessment model, data cleaning must be performed first to remove data noise. In our research, the code and comments contain various identifiers and symbols, such as operators and brackets, which may not frequently appear in other types of plain text. Therefore, the data cleaning process in this paper includes the filtering of special text symbols, word segmentation and

TABLE 1. Details of the labeled dataset. # Tokens refers to the total number of symbols in a corpus, including tokens, operators, and punctuations. # Vocabulary refers to the number of distinct tokens in a corpus. # Average length refers to the ratio of tokens to the category number.

Category	Number	Comment			Code		
		#Vocabulary	#Tokens	#Average Length	#Vocabulary	#Tokens	#Average Length
Coherent	5510	8.4k	83.3k	12.3	27.9k	168.9k	71.3
Non-Coherent	572	3.6k	8.6k	10.8	13.4k	17.2k	66.2

vocabulary processing, in addition to the general data preprocessing techniques conducted for the other text.

We summarize five rules for refining the text information according to the forms of symbols: 1) Remove the inline comments with "//", "/∗", or "∗/" in the code. 2) Remove all Javadoc in comments, such as "@author", "@param", "@deprecated", etc. 3) Remove the contents in comments like "Class#method". These symbols usually refer to methods in other classes that are beyond the scope of the method-level code and comments in this paper. 4) Remove all pairs of brackets and the content within them from the comments, because they usually refer to the explanation of words or phrases and are not helpful to the overall understanding of the comments. 5) Keep punctuations such as "{", "}", and ";", which contain detailed structural information.

We tokenize the identifiers appearing in the code and comments according to word formation (e.g. snake and camel case). We also remove the keywords that appear in the code apart from general stop words, such as "public", "final", and "static", because these words have no real meaning.

D. OUR DATASET

After data cleaning, we build a labeled dataset with 5510 coherent cases and 572 non-coherent cases. Some descriptive statistics of our dataset is reported in Table 1. We take the coherent case as positive and the non-coherent case as negative.

Table 1 suggests that the number of positive samples in the dataset is approximately ten times that of negative samples. In the data filtering process, we choose longer comments to ensure that they are informative. The negative samples are mainly produced by coding mistakes, copy and paste errors, and outdated problem during code evolution. The imbalance of the positive and negative samples poses a challenge to the classification of comments that we should deal with in our framework. And the number of the vocabulary, including some domain-specific words and identifiers, is very large, which increase the difficulty of learning. Finally, for the convenience of future research, we make our dataset publicly available.

IV. FRAMEWORK OF DCOMMENT

To judge the quality of the comments, we propose a comment assessment framework *DComment*, as shown in Fig. 2. Given a corpus of the collected code and comments, *DComment* vectorizes the code and comments after data preparation, during which the attention mechanism is applied to extract important information from the code. We then concatenate the code vectors and the comment vectors and feed them into the MLP classifier for classification. The following subsections describe the detailed design of the framework.

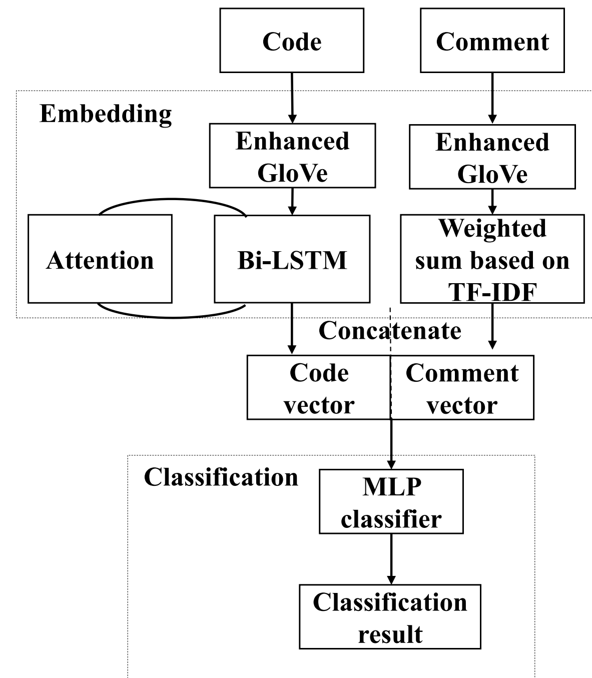


FIGURE 2. An overview of our framework DComment.

A. CODE EMBEDDING

1) THE ENHANCED GLOVE EMBEDDING

Distributed representations [26] have excellent performance in many tasks, such as sentiment classification, similarity evaluation, and reading comprehension. To obtain numeric representations from raw input, we adopt an unsupervised learning algorithm GloVe [22] to learn the representation of the code and comments. GloVe maps tokens into a meaningful space where the distance between tokens is related to the semantic similarity. However, GloVe is a general-purpose vector representation of tokens trained on enormous general datasets.

To get the fine-tuned representations for the code and comments, we introduce a simple retrofitting-like extension to the original GloVe model [23]. We fine-tune the general representations on a large corpus of the Java projects from GitHub, so that we can synthesize the domain-specific knowledge and

the general-purpose representations in a way that both representations of the code and comments as different language types are proper and informative.

2) THE Bi-LSTM NEURAL NETWORK

The code contains low-level implementation details using many tokens to implement the task described by the comment and therefore the information of the code is very sparse. To fully exploit the sparse information, we apply the Bidirectional LSTM (Bi-LSTM). Bi-LSTM contains two subnetworks for the left and right sequence context, which can capture bilateral semantic dependence in the code. Given a code snippet S, where $S = \{v_1, v_2, \ldots, v_{Ns}\}$ is the vector representation of N_S tokens in the code snippet, the hidden state can be represented as the following equations:

$$\vec{h}_t = tanh\left(\vec{W}\left[\vec{h}_{t-1}; v_t\right]\right), \quad \forall t = 1, \ldots, N_s \tag{2}$$

$$\overleftarrow{h}_t = tanh\left(\overleftarrow{W}\left[\overleftarrow{h}_{t+1}; v_t\right]\right), \quad \forall t = N_s, \ldots, 1 \tag{3}$$

where $v_t \in R^d$ is the vector representation of token t, $tanh$ is the activation function of the Bi-LSTM, $\vec{W}$ and $\overleftarrow{W} \in R^{2d \times d}$ are the matrices of the trainable parameters of the forward and backward pass, respectively. We concatenate $\vec{h}_t$ and $\overleftarrow{h}_t$ to get a representation of the code:

$$\mathrm{h}_t = \left[\vec{h}_t; \overleftarrow{h}_t\right], \quad \forall t = 1, 2, \ldots, N_s \tag{4}$$

h_t summarizes the forward and backward sequence information around token t.

3) ATTENTION MECHANISM for code embedding

However, not all tokens contribute equally to the representation of the code. Hence, we introduce the attention mechanism [24] to extract important tokens and aggregate them to represent the code.

According to (4), we get a matrix H, where $H = [h_1, h_2, \ldots, h_{N_s}]$ is the output vectors of Bi-LSTM. Then we randomly initialize a vector that will be fine-tuned in the training process and use the softmax function to get the weight vector as shown in (6). The weight vector represents the importance of each token in the code. We can get the weighted sum as the code representation r_S by the following expressions:

$$M = tanh\,(H) \tag{5}$$

$$\alpha = softmax\left(W^T M\right) \tag{6}$$

$$r_S = H\alpha^T \tag{7}$$

where $H \in R^{d \times N_s}$, $W \in R^d$ is the trained parameter vector and W^T is the transpose of W. α is the weighting factor of H and its dimension is N_s.

B. COMMENT EMBEDDING

Comments provide a high-level description of the tasks performed by the code. The tokens in comments are informative and too much processing will lead to the loss of information. Similar to the code embedding, we apply the enhanced GloVe method to embed the tokens in the comments, and there is a corresponding vector for each token. For a comment with a sequence of tokens, previous studies usually use the sum or the average of the token vectors to represent the entire sequence.

In our work, we assign different weights to each token vector according to the relevance of tokens with the corresponding code. We implement this idea through TF-IDF [25].

Given a data collection D, a comment token t_i, and the corresponding code snippet c, we calculate the weight of t_i in as follows:

$$w_{t_i} = f_{t_i,c} \times \log \frac{|D|}{f_{t_i,D}} \tag{8}$$

where $f_{t_i,c}$ is the number of times that t_i appears in c, $|D|$ is the size of the corpus, and $f_{t_i,D}$ equals the number of code in which t_i appears in D.

The representation of the comment r_C can be calculated in the following equation:

$$r_c = \sum_{i=1}^{N_T} w_{ti} t_i \tag{9}$$

where $\boldsymbol{N_T}$ is the number of the tokens in the comment.

C. CLASSIFICATION

Once we get the representation of the code snippet r_S and the comment r_C, we concatenate them as the representation $r = [r_S; r_C]$. To map the distributed representation to the sample space and find the correlation between the code and the comment, we feed the vectors of the representation r into the MLP classifier and use the sigmoid function to get the classification results. The loss is calculated by the binary cross-entropy function, which measures the performance of a classification model whose output is a probability value between 0 and 1:

$$\mathrm{L}\,(\theta) = -\left[y_i \log \hat{y}_i + (1 - y_i) \log\left(1 - \hat{y}_i\right)\right] \tag{10}$$

where θ denotes the model parameters, y_i is the real label for the i-th class, $\hat{y}_i$ is the prediction probability for the i-th class, and $y_i, \hat{y}_i \in [0, 1]$. The value of 0 represents the positive example, and the value of 1 represents the negative sample.

D. DATA IMBALANCE PROCESSING

The number of the positive samples and that of the negative samples are seriously unbalanced, as shown in Table 1. To balance the input for our framework, we introduce the class weight mechanism, by which samples within the class with smaller amount will be assigned for higher penalty weights, and vice versa. We set the higher penalty weight $\varepsilon = 0.75$ to the negative sample class, and the lower penalty weight $1 - \varepsilon = 0.25$ to the positive sample class, based on the ratio of the number of the positive and negative samples. The loss function then becomes a weighted average, where the weight of each sample is specified by the penalty weight of the class it belongs to.

TABLE 2. Details of the datasets using for training and testing.

Dataset	Positive samples	Negative samples	Training data		Testing data	
			Positive samples	Negative samples	Positive samples	Negative samples
Our dataset (After resampling)	5510	572	4752	1584	1102	114
Corazza *et al.* [9]	2468	1168	1974	934	494	234

TABLE 3. Overall Comparison on two datasets.

Model	Our dataset			Corazza *et al.* [9]		
	Precision	Recall	F1-score	Precision	Recall	F1-score
SVC	89.60%	93.58%	91.55%	83.50%	90.78%	86.99%
TBM	90.14%	92.68%	91.39%	87.20%	89.51%	88.34%
GCM	**97.71%**	92.15%	94.85%	**90.13%**	92.19%	91.15%
GBM	93.64%	95.90%	94.76%	88.44%	92.00%	90.18%
DComment	96.99%	**96.84%**	**96.91%**	89.71%	**94.21%**	**91.90%**

We also apply the random under-sampling and over-sampling to the training set. After resampling, the ratio of the number of the selected positive samples to that of the selected negative samples in the training set changes from 9.63:1 to 2.55:1, as shown in Table 2.

V. EXPERIMENTS AND RESULTS

In this section, we conduct experiments to evaluate the performance of our proposed *DComment* framework.

A. DATASETS

We include two datasets in our experiments. One is our manually labeled dataset from GitHub and the other is a public dataset provided by Corazza *et al.* [9]. The public dataset includes the results of a manual assessment on the coherence between comments and the implementations of 3636 methods, gathered from three open source softwares implemented in Java. We randomly select 20% samples for each dataset as its testing set, and the rest serves as its training set. We enhance our training set by resampling. The details of the datasets are listed in Table 2.

B. EVALUATION METRICS AND BASELINES

We adopt three classical metrics, namely precision, recall, and F1-score [29], for the measurement of the comment classification. We compare *DComment* with the state-of-the-art work [9]. To validate the effectiveness of our comment classification framework, we also compare *DComment* with some variations, in which we replace the components of *DComment* with other deep learning methods and embedding methods.

We list the evaluation baselines as follows:

- SVC: The Support Vector Classification is used in the work of Corazza *et al*., and Radial Basis Function Gaussian kernel is set.
- TBM (TF-IDF + Bi-LSTM + MLP): To evaluate the effectiveness of the pretrained embedding, we use TF-IDF to vectorize the code and comments instead of GloVe.
- GCM (GloVe + CNN + MLP): We compare Bi-LSTM with CNN in code embedding, which has achieved excellent performance in text processing [27], [28].
- GBM (GloVe + Bi-LSTM + MLP): We adapt *DComment* to apply Bi-LSTM for both the code and comments, to verify the intuition that treating the code and comments in different ways is more suitable for our task.

C. EXPERIMENTAL CONFIGURATION

For the embedding process, we set the vector length of the code and the comments to 160 and 16, respectively. In the training process, we set the number of Bi-LSTM hidden units to 128. The batch size is set to 32 and the number of epochs is set to 50. We select the sigmoid function as the activation function and RMSProp as the optimizer. The learning rate is set to 0.25.

D. EXPERIMENTAL RESULTS

1) COMPARISON WITH BASELINES

We compare the performance of *DComment* and the baselines. As shown in Table 3, *DComment* outperforms the other approaches in terms of recall and F1-score. Although *DComment* is slightly lower in accuracy than GCM, *DComment* outperforms these baselines in most cases and is stable on distinct metrics. Our approach gets the best recall performance of 96.84% on our labeled dataset, which outperforms the previous work SVC, TBM, and GCM, by the value of 3.26%, 4.16%, and 4.69%, respectively. *DComment* achieves the best F1-score of 96.91%, which is slightly lower than that of GCM, but it is still comparable. The findings on the dataset provided by Corazza *et al.*[9] are similar, and *DComment* is still the overall winner in performance. The recall and F1-score of *DComment* reach to 89.71% and 94.21%, respectively.

According to the comparison of TBM and *DComment*, we can observe that GloVe performs better than TF-IDF in vector representation, which indicates that taking the contextual semantics into consideration may be better than only considering word frequency in vector representation.

The performance of SVC is lower than the other approaches in most metrics. In these baselines, SVC is the only approach that uses the traditional machine learning classifier for comment classification, and the others apply deep neural network models in classification. It indicates the potential advantage of deep neural network models in comment classification, compared to traditional machine learning based methods.

Recently, some researchers have introduced CNN into the NLP field [27], [28] for classification and other tasks. In our experiments, GCM, which uses CNN as a component for code embedding, obtains relatively good performance, especially in the precision metrics. This finding shows that CNN may have advantages in extracting latent features and text classification. However, the overall performance of GCM is still lower than that of our approach.

We also find that *DComment* performs better than GBM over all metrics. The main difference between GBM and *DComment* is that GBM processes the code and comments in the same way, using the general Bi-LSTM model. The improvement in the performance of *DComment* suggests that treating the code and comments differently with the consideration of their different characteristics may be a sound way to better accomplish the comment classification task.

2) IMPACT OF PRE-TRAINING STRATEGIES

This section investigates whether different pre-training strategies affect the overall performance of *DComment* and the baselines. We train the global vectors for token representation with two pre-training strategies, respectively. For the general-purpose pre-training strategy, we use the general pre-trained vectors [22] as the global vectors. For the domain-specific pre-training strategy, we tune the general pre-trained vectors on a large corpus of Java projects from GitHub utilizing Mittens [23].

Fig. 3 shows the performance of GCM, GBM, and *DComment* with different pre-training strategies. The performance differences between general purpose and domain-specific pre-training strategies are minimal over three metrics. Overall, approaches with domain-specific pre-training strategy slightly outperform approaches with general purpose pre-training strategy and the improvement is limited considering the cost of obtaining a corpus and computing embeddings.

3) THE GENERALIZATION ABILITY OF DCOMMENT

Moreover, we assess the generalization ability of *DComment*.

We randomly split our labeled dataset by 4:1 into the training set and the testing set. Then we train *DComment* using the training set and the whole public dataset [9] to obtain two classifiers. Finally, we use the testing set from our labeled dataset to compare the performance of the two classifiers trained on different training sets.

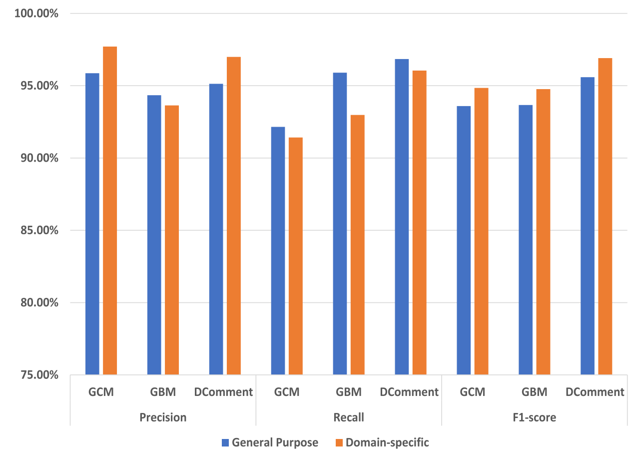


FIGURE 3. Comparison of different approaches with different pre-training strategies.

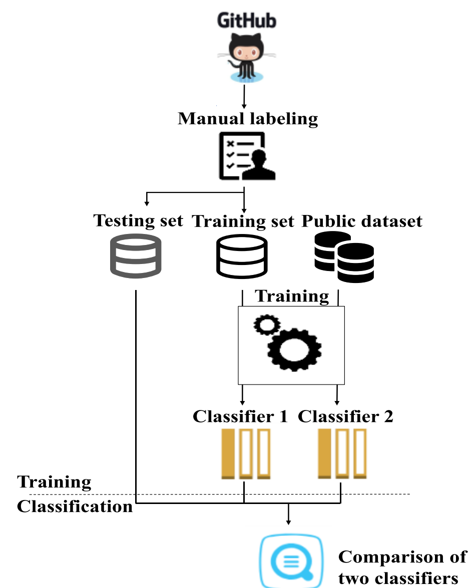


FIGURE 4. The validation method of the generalization ability.

Fig. 4 shows an overview of our validation method. We observe that F1-score drops from 96.91% to 85.80% when we change the training set from our dataset to the public dataset. A possible reason is that there are intent factors such as code style that affects the performance of *DComment*. When we train and test *DComment* on the same dataset, *DComment* is easy to learn special features unique to the dataset, while these features may lead to overfitting. However, our approach still reaches high performance in the experiment of different datasets, indicating high generalization ability over data.

4) ATTENTION VISUALIZATION

In this section, we validate whether the attention mechanism works in our framework *DComment*. We choose a coherent

[2]It is available at agile.csc.ncsu.edu/SEMaterials/tutorials/coffee maker/.

TABLE 4. An example of the coherent code-comment pair.[2]

CoffeeMaker_Web/src/edu/ncsu/csc326/coffeemaker/Main.java, 273, 287

Comment:

```
/**
 * Passes a prompt to the user that deals with the recipe list
 * and returns the user selected number.
 * @param message
 * @return int
 */
```

Code:

```
private static int recipeListSelection (String message) {
  String userSelection = inputOutput(message);
  int recipe = 0;
  try {
    recipe = Integer.parseInt(userSelection) - 1;
    if (recipe >= 0 && recipe <=2) {
      //do nothing here.
    } else {
      recipe = -1;
    }
  } catch (NumberFormatException e) {
    System.out.println("Please select a number from 1-3.");
    recipe = -1;
  }
  return recipe;
}
```

Code input after cleaning: recipe select message user select message recipe parse user select recipe format system select recipe (Ignoring the punctuation)

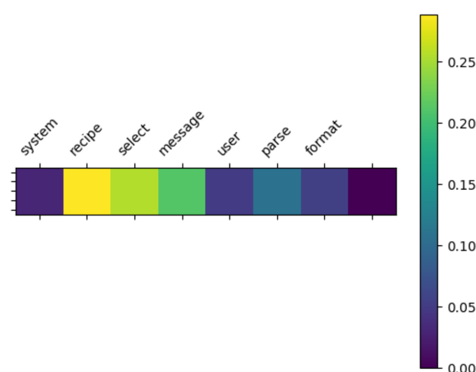


FIGURE 5. The visualization of attention weights on tokens in the code input.

example of the code-comment pair as shown in Table 4. The raw code and comment of the method is in the top half of the table, and the code after data cleaning is shown in the bottom. For simplification, we ignore punctuations in this example.

Fig. 5 shows the attention weights of tokens in the code input. The tokens with higher weights are more important and informative. In this example, "recipe" and "select" are given higher weights. On the contrary, the weights of "system", "user" and "format" are close to zero. According to the context in Table 4, we find that the word " recipe" and " select" are meaningful indeed, and are closely related to the task the comment describes. However, the word "system" and "format" are meaningless and cannot help understand the important tokens and embed informative code vectors. The experiment suggests that the attention mechanism can help us capture the relevance between the code and the comment and embed informative code vectors.

E. COMPARISON WITH GENERAL SENTENCE CLASSIFICATION

Although the code and comments are significantly different from plain text, sentence classification for general texts may be used for comment assessment. To illustrate its performance, we run the state-of-the-art sentence classification model SciBERT [39] on the public dataset. The data, model and results are publicly available.[3]

The accuracy and F1-score of SciBERT are 83.33% and 80.96%, respectively, which are significantly lower than the results of our model *DComment*. It does not perform well in the area of comment assessment. According to the comparison of SciBERT and our model, specific data processing techniques and model architecture design for the code and comments seem to be necessary.

VI. CONCLUSION

In this paper, we describe a multi-input neural network framework *DComment* for the automatic comment assessment. *DComment* vectorizes the tokens in the code and comments using the enhanced GloVe model, which is fine-tuned on the domain-specific corpus. We train a Bi-LSTM model to summarize the token vectors of code and introduce the attention mechanism to focus on the tokens with important information. For comments, we sum up the token vectors as the comment representations, with the consideration of token weights, which are calculated based on TF-IDF. Finally, we feed the code vectors and the comment vectors into the MLP classifier. Experiments on two datasets of code-comment pairs show that *DComment* outperforms the previous work and several variations of deep neural networks significantly by adapting the model to different characteristics of the code and comments, and has high generalization performance over data from different sources.

For the future, we plan to include more information in *DComment*, such as API patterns and improve the generalization ability of *DComment* for different types of languages. Our ultimate goal is to create a large, public, and high-quality code-comment corpus supporting software engineering tasks such as code summarization, code completion, and comment generation.

REFERENCES

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018.

[2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proc. 5th Int. Conf. Softw. Eng.*, Mar. 1981, pp. 215–223.

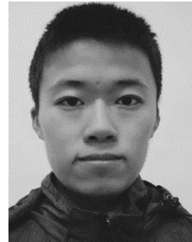
[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Annu. Int. Conf. Design Commun.*, 2005, pp. 68–75.

[4] B. P. Lientz, "Issues in software maintenance," *ACM Comput. Surv.*, vol. 15, no. 3, pp. 271–278, Sep. 1983.

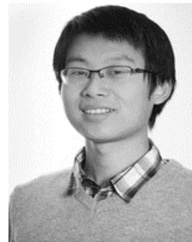
[5] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 83–92.

[3] It is available at https://github.com/wangdeze18/Validation-by-Scibert

[6] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers—Taxonomies and characteristics of comments in operating system code," in *Proc. 31st Int. Conf. Softw. Eng.*, May 2009, pp. 331–341.
[7] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *Proc. 27th Int. Conf. Program Comprehension (ICPC)*, 2019, pp. 53–64.
[8] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The javadocminer," in *Proc. 10th Int. Conf. Appl. Natural Lang. Inf. Syst.*, 2010, pp. 68–79.
[9] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: A dataset and an empirical investigation," *Softw. Qual. J.*, vol. 26, no. 2, pp. 751–777, Jun. 2018.
[10] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *Proc. 42nd Annu. Comput. Softw. Appli. Conf. (COMPSAC)*, vol. 1, Jul. 2018, pp. 154–163.
[11] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "Comment: Testing javadoc comments to detect comment-code inconsistencies," in *Proc. 5th Int. Conf. Softw. Test. Verification Validation*, Apr. 2012, pp. 260–269.
[12] D. J. Lawrie, H. Feild, and D. Binkley, "Leveraged quality assessment using information retrieval techniques," in *Proc. 14th Int. Conf. Program Comprehension (ICPC)*, Jun. 2006, pp. 149–158.
[13] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 227–237.
[14] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On end-to-end program generation from user intention by deep neural networks," 2015, *arXiv:1510.07211*. [Online]. Available: https://arxiv.org/abs/1510.07211
[15] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Int. Conf. Mining Softw. Repositories (MSR)*, 2015, pp. 334–345.
[16] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, May/Jun. 2018, pp. 933–944.
[17] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. Int. Conf. Softw. Maintenance Evol.*, Sep. 2017, pp. 115–124.
[18] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, May/Jun. 2018, pp. 476–486.
[19] D. Fucci, A. Mollaalizadehbahnemiri, and W. Maalej, "On using machine learning to identify knowledge in API reference documentation," 2019, *arXiv:1907.09807*. [Online]. Available: https://arxiv.org/abs/1907.09807
[20] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: The kappa statistic," *Family Med.*, vol. 37, no. 5, pp. 360–363, 2005.
[21] J. L. Fleiss. (1981). *Statistical Methods for Rates and Proportions*. Assessed: Oct. 30, 2019. [Online]. Available: https://onlinelibaray.wiley.com/doi/pdf/10.1002/0471445428.fmatter
[22] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1532–1543.
[23] N. Dingwall and C. Potts, "Mittens: An extension of GloVe for learning domain-specialized representations," 2018, *arXiv:1803.09901*. [Online]. Available: https://arxiv.org/abs/1803.09901.
[24] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-based bidirectional long short-term memory networks for relation classification," in *Proc. 54th Assoc. Comput. Linguistics*, 2016, pp. 207–212.
[25] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, 1988.
[26] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, "Distributed representations," in *Parallel Distributed Processing*, vol. 1. Cambridge, MA, USA: MIT Press, 1986.
[27] C. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts," in *Proc. 25th Int. Conf. Comput. Linguistics*, Aug. 2014, pp. 69–78.
[28] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Oct. 2014, pp. 1746–1751.
[29] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, 2008. [Online]. Available: https://nlp.stanford.edu/IR-book/information-retrieval-book.html
[30] Y. Shinyama, Y. Arahori, and K. Gondow, "Analyzing code comments to boost program comprehension," in *Proc. 25th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2018, pp. 325–334.
[31] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, vol. 7, pp. 111411–111428, 2019.
[32] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. (SANER)*, Mar. 2015, pp. 380–389.
[33] H. Yu, B. Li, P. Wang, D. Jia, and Y. Wang, "Source code comments quality assessment method based on aggregation of classification algorithms," *J. Comput. Appl.*, vol. 36, no. 12, pp. 3448–3453, 2016.
[34] I. Beltagy, K. Lo, and A. Cohan, "SciBERT: A pretrained language model for scientific text," 2019, *arXiv:1903.10676*. [Online]. Available: https://arxiv.org/abs/1903.10676



DEZE WANG received the B.S. degree from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2018, where he is currently pursuing the Ph.D. degree. His main research interests include machine learning and text representation.



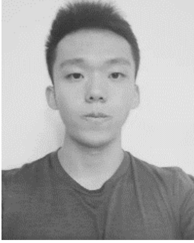
YONG GUO received the B.S. degree in computer science and technology from the National University of Defense Technology, Changsha, China, in 2009, and the Ph.D. degree in computer science from the Delft University of Technology, the Netherlands, in 2016.

He is currently a Lecturer with the College of Systems Engineering, National University of Defense Technology. His research interests are in distributed computing systems, large-scale graph processing, and deep learning on graphs.

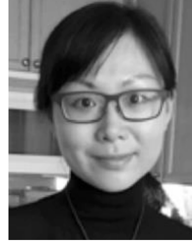
WEI DONG received the B.S. and Ph.D. degrees in computer science from the National University of Defense Technology, Changsha, Hunan, China, in 1997 and 2002, respectively.

He was a Lecturer, from 2002 to 2004 and an Associate Professor, from 2004 to 2010 with the College of Computer Science, National University of Defense Technology, where he has been the Professor, since 2010. He has authored more than 60 articles and two textbooks. He has served on more than 20 program committees. His research interests include runtime verification, software analysis and testing, model checking, and intelligent software development. He has served as the Program Co-Chair for several conferences and workshops.

ZHIMING WANG received the B.S. degree from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2018. His main researches include software engineering and deep learning.



HAORAN LIU received the B.S. degree from the College of Computer Science, National University of Defense Technology, Changsha, China, in 2018, where he is currently pursuing the M.S. degree. His main research includes software engineering.



SHANSHAN LI received the M.S. and Ph.D. degrees from the College of Computer Science, National University of Defense Technology, in 2003 and 2007, respectively. She was a Visiting Scholar with The Hong Kong University of Science and Technology, in 2007, and also with the University of Minnesota, in 2014.

She is currently a Full Professor with the Department of Computer Science, National University of Defense Technology. She is also the Leader of the LEOPARD Laboratory. Her main research areas include empirical software engineering and intelligent software development, with a particular interest in software quality enhancement, defect prediction, and misconfiguration diagnosis. She has published more than 50 articles and received several awards including the FSE, ASE, ICPC, SANER, and TPDS.