The Journal of Systems and Software 165 (2020) 110572

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Improving software bug-specific named entity recognition with deep neural network

Cheng Zhou[a,b], Bin Li[a,d,*], Xiaobing Sun[a,c,*]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] Taizhou University, Taizhou, China
[c] Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology, Nanjing, China
[d] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

ARTICLE INFO

Article history:
Received 24 July 2019
Revised 6 January 2020
Accepted 9 March 2020
Available online 10 March 2020

Keywords:
Software bug analysis
Named entity recognition
Software bug corpus
LSTM-CRF

ABSTRACT

There is a large volume of bug data in the bug repository, which contains rich bug information. Existing studies on bug data mining mainly rely on using information retrieval (IR) technology to search relevant historical bug reports. These studies basically treat a bug report as a closed unit, ignoring the semantic and structural information within it. Named-entity recognition (NER) is an important task of information extraction (IE) technology. Based on NER, fine-grained factual information could be comprehensively extracted to further form structured data, which provides a new way to improve the accessibility of bug information. However, bug NER is different from general NER tasks. Bug reports are free-form text, which include a mixed language environment studded with code, abbreviations and software-specific vocabularies. In this paper, we propose a deep neural network approach for bug-specific entity recognition called DBNER using bidirectional long short-term memory (LSTM) with Conditional Random Fields decoding model (CRF). DBNER extracts multiple features from the massive bug data and uses attention mechanism to improve the consistency of entity tags in the bug reports. Experiment results show that the F1-score reaches an average of 91.19%. In addition, in cross-project experiments, the DBNER's F1-score reaches an average of 84%.

1. Introduction

Software bug issues are unavoidable in software development and maintenance. In order to facilitate management of these software bugs, many large software systems are equipped with special bug tracking systems such as Bugzilla[1] to collect and track the bugs of software projects. In a bug tracking system (BTS) (also known as a bug repository), the entire life cycle of each bug is well tracked in the form of a bug report, from being submitted or opened to being fixed or reviewed (Sun et al., 2017b). Fig. 1 shows a bug report from the Mozilla project.

There are many studies dedicated to mining historical bug reports to help fix new bugs, such as bug prediction (D'Ambros et al., 2010; Herzig et al., 2013), bug location (Chaparro, 2017; Shokripour et al., 2013), bug assignment (Jonsson et al., 2016; Sun et al., 2018; Yang et al., 2016a) and bug categorization (Tan et al., 2014; Schuh and Slater, 1995). These studies often treat bug reports as unstructured textual documents and use information retrieval (IR) techniques to find historical bug reports similar to the given new bug, such as vector space model (e.g., TF-IDF(Yang et al., 2016b)), topic model (e.g., LDA (Blei et al., 2003; Sun et al., 2016)) or neural network language model (e.g., word embedding (Zhou et al., 2015)). However, existing studies have some limitations on mining bug data. First, the overall structural features and inherent semantic information of the bug report are ignored during the data processing. Taking bug 1,494,924 in Fig. 1 as an example, after natural language processing, its title is decomposed into ten unrelated tokens (i.e., bad, performance, google, docs, cause, fallback, blob, image, svg and filter). Phrases with intrinsic correlation such as "svg filter" are broken down. The sibling relationship between the items fallback, blob image, and svg filters has also been ignored, not to mention the cause information of the bug that these three items have clearly indicated. Secondly, new bugs often contain new words that appear less frequently or have never appeared in historical bug data. These words have important meanings, but are ignored during the calculation process because they cannot be

* Corresponding authors.
E-mail addresses: lb@yzu.edu.cn (B. Li), xbsun@yzu.edu.cn (X. Sun).
[1] https://www.bugzilla.org/.

https://doi.org/10.1016/j.jss.2020.110572
0164-1212/

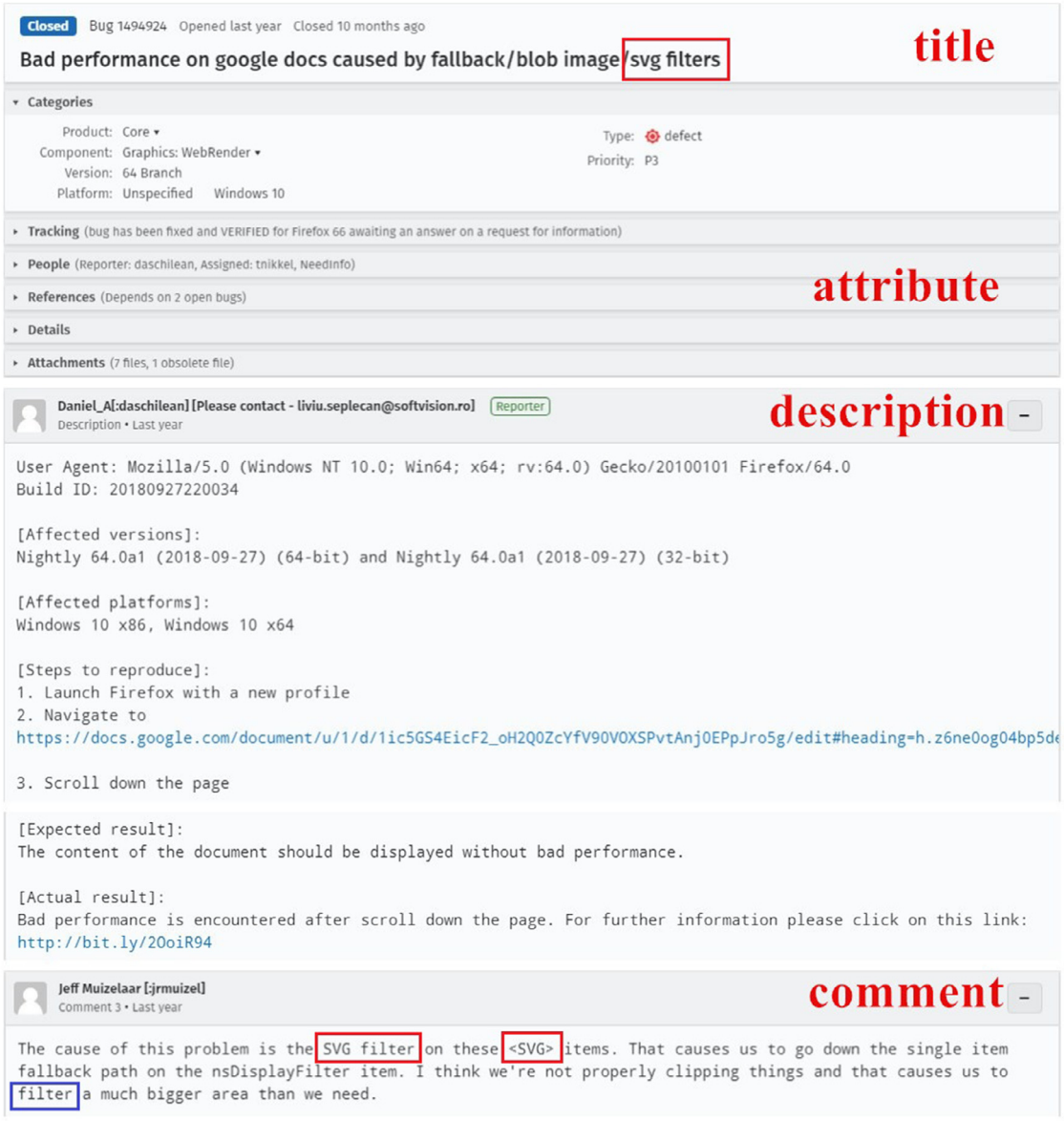

Fig. 1. Bug 1,494,924 in Mozilla's bug tracking system.

matched. Finally, after being retrieved from the bug repository, developers still have to spend a lot of time reviewing and analyzing these bug reports in order to extract the information they want. These limitations will reduce the accuracy of information retrieval and affect the efficiency of bug repair tasks.

Named-entity recognition (NER) (also known as entity identification, entity chunking and entity extraction) is a subtask of information extraction that seeks to locate and classify named entity mentions in unstructured text into predefined categories such as the person names, organizations, locations, medical codes, etc. (McCallum and Li, 2003). Named entities could be words or phrases composed of words, with flexible forms. Based on NER, fine-grained factual information such as relations between entities and events could be comprehensively extracted to further form structured data, which provides an new way for the representation, management, and efficient reuse of massive bug data. NER is mainly used in the fields of knowledge graph (Yan et al., 2018), machine translation (Hassan et al., 2016), question answering system, and has been deeply studied in many domains such as news, medical, finance, etc. However, there are still few studies in the software bug domain. The software bug-specific named entity recognition task faces many challenges:

- There is no proper classification method for bug-specific named entity and few standard corpus in software bug domain. The accurate labeled corpus is one of the most important inputs for the NER system.
- For a software system, one of its characteristics is its evolvability, that is, it keeps evolving to enhance various users' feature requests, fix bugs, etc. Hence, the baseline bug corpus needs to consider such evolvability characteristic to better support the NER task.
- When reporting a bug report, the users do not always describe bugs following the standardized naming rules or description formats. Moreover, the bug data usually includes the text composed of a mixed language studded with code, abbreviations, software specific vocabularies and vague language, etc., which makes the NER more difficult.
- The bug NER task is performed on a bug report, not just a sentence. The current methods (Zhou et al., 2018; Ma and

Hovy, 2016; Chen et al., 2019), whether based on traditional machine learning methods or deep learning methods, do not make good use of document level information, just focus on the sentence level. This creates the inconsistency of the entity tags (the same entity in a bug report is labeled as different category tags).

In this paper, by investigating into a large number of bug reports in software bug repository, three characteristics of entities in bug reports are summarized: variety parts of speech (POS), description phrases, and solid distribution. Based on these characteristics, we propose a category method for bug data, and build a baseline bug data corpus on four open source projects, Mozilla[2], Eclipse[3], Apache[4] and Kernel[5]. Then, using the corpus as training set and test set, we propose a novel neural network based approach named *DBNER* for the bug-specific named entity recognition. We extend the bidirectional long short-term memory (BiLSTM) with Conditional Random Fields decoding (CRF) method (Huang et al., 2015; Lample et al., 2016a), which has been successfully used in various sequence labeling tasks, as the basic architecture. LSTM is suitable for handling multiple inputs and has a long-term memory. BiLSTM can more effectively capture the features of the context to identify named entities based on sentence level information. Due to the mixed language environment of bug reports, we combine the orthographic feature, embedding feature, and domain features (i.e., POS feature and gazetteer feature[6]) as the input. Moreover, since the object of NER task is a bug document composed of multiple sentences, we introduce the attention mechanism to obtain the context information of entities at the document level, so as to solve the problem of inconsistent tags of the same entity in different sentences. Finally, we conduct an empirical study on the four software projects to evaluate our *DBNER* system. Compared with basic BiLSTM-CRF model, the F1-score increases by 2.5% on average after applying the attention mechanism, and 4% on average after further introduction of domain features. Using attention mechanism and domain features both enhance model performance. We perform two cross-project NER experiments in an unsupervised fashion, and the F1-score reaches 84% on average.

This article extends our preliminary study published as a research paper in a conference (Zhou et al., 2018). In our preliminary study, we explored the characteristics of bug data and bug-specific named entities, defined the bug entity classification criteria, built a corpus consists of two software projects (i.e., Mozilla and Eclipse), and proposed a approach based on machine learning method, which provided a baseline. This article extends the preliminary study in various ways:

- The basic corpus consists of more software projects, including both application software projects (i.e., Mozilla, Eclipse and Apache) and a system software project (i.e., Kernel), which is more comprehensive.
- Compared with previous approach *BNER* using CRF model, *DBNER* adopt a deep learning method (attention-based BiLSTM-CRF), which could automatically mine multiple features from word-level, sentence-level and document-level at the same time.
- A wider cross-projects experiment with more subjects and more cross-validation methods, is performed to fully evaluate the proposed *DBNER* approach.

The major contributions of our work are shown as follows:

- We propose a software bug-specific NER approach called *DBNER*, using deep neural network architectures (attention-based BiLSTM-CRF) combining with domain features (i.e., POS feature and gazetteer feature), which can effectively help explore semantic information in bug data with little feature engineering based on the baseline corpus.
- We evaluate *DBNER* on four open source projects, including both application and system software projects. Compared to our previous method *BENR*, in the individual-project experiment, the precision rate of *DBNER* has increased by an average of 4.5%, the recall rate has increased by an average of 4.15%, and the average F1-score has increased by 4.07%. In addition, in cross-projects experiments, the *DBNER's* F1-score has increased by an average of 5.8%. The results demonstrate that the designed baseline corpus is suitable for bug-specific named entity recognition, and the *DBNER* approach using a generalizable model shows better performance for both individual-project and cross-projects NER task.

The rest of this paper is organized as follows. In Section 2 we discuss prior work and how it relates to our approach. The preliminaries are introduced in Section 3. We present our approach to implement NER in Section 4. Section 5 shows our empirical study. We discuss threats to validity in Section 6. Finally, Section 7 concludes this paper and discusses the future work.

2. Related work

In this section, we first highlight the most related work of NER in the software domain in section 2.1. Then, we present other work about software information extraction in Section 2.2.

2.1. Named entity recognition in software domain

In natural language text, entity is the basic information element, which usually indicates the main content of the text. Named entity recognition, *NER*, the problem of identifying the underlying entities from the data, has been studied for many decades in many communities (Sang, 2002a; Chinchor, 1998). It first appeared in MUC-6 (message understanding conferences), which focused on information extraction. There are usually two methods to solve the NER problem, one is to use linear statistical model, such as support vector machine (SVM), hidden Markov model (HMM) and conditional random field (CRF). All these models need various feature templates, such as the grammatical features of words. Although feature templates can accurately obtain valuable feature information and make up for the lack of corpus labeling, these models have poor generalization capabilities because different domain data sets require different feature templates. The other method is deep learning. Collobert is the first to use deep neural network for named entity research (Collobert and Weston, 2008). Deep learning methods treat NER as a sequential annotation task. At present, researchers mainly adopt LSTM architecture with traditional CRF model, and use pre-trained word embedding as an additional feature to solve the NER problem (Huang et al., 2015; Lample et al., 2016b; Ma and Hovy, 2016; Luo et al., 2018). Deep learning methods do not need to manually set feature templates, and can automatically mine contextual information in text.

However, in the field of software, NER has not been extensively studied. Mahalakshmi et al. focused on automating the test process from the early stages of requirement elicitation in the development of software, and described a semi-supervised NER technique to generate test cases in the given set of use cases (Mahalakshmi et al., 2018). Ye et al. proposed an S-NER system using CRF model and Brown clustering to recognize software-specific entities such as programming languages, platforms and libraries on

[2] https://bugzilla.mozilla.org/describecomponents.cgi.
[3] https://bugs.eclipse.org/bugs/.
[4] https://bz.apache.org/bugzilla/.
[5] https://bugzilla.kernel.org/query.cgi.
[6] We build a software bug dictionary, which is flexibly further divided into gazetteers corresponding to the defined entity categories as shown in Table 2

Stackoverflow posts to extract software knowledge from community documents (Ye et al., 2016). Lin et al. defined the types and attributes of entities in different software artifacts, and proposed a joint knowledge graph application framework to provide intelligent assistance in the life cycle of software development (Lin et al., 2017). Li et al. extracted ten subcategories of API caveat sentences from API documentation and linked these sentences to the API entities in an API knowledge graph, to improve API recommendation work and dig deeper concepts and API knowledge (Li et al., 2018). In our previous research, we defined the categories and attributes of bug-specific named entities, and proposed a CRF-based method BNER to identify entities in bug reports (Zhou et al., 2018; Zhou, 2018). Like other NER research goals in the software field, we are committed to freeing up information in bug relevant software artifacts, which are aggregated, mined and analyzed into structured data to improve the accessibility of bug knowledge in bug documents.

2.2. Information extraction in software domain

There has been much research focusing on software information extraction. Software big data exist in various forms, including documents, SCM documents, source code, bug data and mailing lists (Sun et al., 2015; Ibrahim et al., 2007). Rigby and Robillard proposed a traceability recovery approach to extract the code elements contained in various documents and showed the notion of code salience as an indicator of the importance of a particular code element (Rigby and Robillard, 2013). Witte et al. mined the software data documents at semantic level and the extracted information was used in automated population of documentation ontology (Witte et al., 2007). Hauff et al. utilized the DBPedia Ontology to extract software concepts from GitHub developer profiles (Hauff and Gousios, 2015). Dagenais and Robillard developed RecoDoc to extract Java APIs from several learning resources (formal API documentation, tutorial, forum posts, code snippets) and then perform traceability link recovery across different sources (Dagenais and Robillard, 2012).

There are also some studies focusing on mining information from bug reports. Shokripour et al. used part-of speech information to find software noun terms in bug reports (Shokripour et al., 2013). Zhang et al. investigated the authorship characteristics of contributors in bug repositories and leveraged the authorship characteristics to resolve software tasks (Zhang et al., 2017). Sun et al. proposed an approach to enhance developer recommendation by extracting expertise and developing habits from historical commits (Sun et al., 2017a). Chaparro proposed an approach to identify, enforce, and leverage the discourse that users utilized to describe software bugs to improve bug reporting, duplicate bug detection and localization (Chaparro, 2017). In our work, we focus on analyzing the bug data to build a baseline bug corpus, which is further used for named entity recognition.

3. Preliminaries

In this section, we describe the preliminary knowledge related to our approach. First, we summarize the characteristics of bug data. Then, we describe the classification criteria for bug-specific named entities. Finally, we introduce the attention-based BiLSTM-CRF model for NER.

3.1. Bug data

Nowadays, bug tracking systems, such as Bugzilla[7], are widely used in software projects to store and manage bug data in the form of bug reports. Along with the development of software projects, these bug tracking systems contain tremendous bug reports. There are currently about 137 companies, organizations, and projects that use Bugzilla to track product bugs listed on the Bugzilla website. According to the statistics, up to March 2019, Bugzilla has managed over 546,000 bug reports for Eclipse and 1,543,000 for Mozilla, respectively. When a bug is found, a bug report is initialized by a user or a tester to describe its details, e.g., symptoms, related attributes and steps to reproduce this bug. In bug tracking systems, a bug could be a defect, a new feature, an update to documentation, or a refactoring (Herzig et al., 2013).

When a user or tester submits a new bug report to the software bug repository, the bug report may include these parts: title, description, comments, attachment, and multiple features such as reporter, assignee (i.e., fixer), component and product as shown in Fig. 1. Among them, the title is a brief description of a bug; the description shows the detailed information of the bug; the comments indicate the free discussion about the reported bug; the component indicates which component is affected by the bug; and the product shows which product is influenced by the bug. To ensure correct entity classification, we only study verified fixed bugs whose root causes can be identified from bug reports and the relevant commit messages, because unfixed bugs may be invalid and the root causes described in the reports may be wrong, thus misleading humans' annotation of entities (Tan et al., 2014).

As shown in Table 3, We collected a total of 175,000 fixed bug reports and extracted the title, description, comments, product and component from each bug report. Then, from them we randomly sampled a diverse set of 1000 bug reports from Mozilla, 400 bug reports from Eclipse, 1000 bug reports from Apache and 200 bug reports from Kernel covering major components of each project to construct our corpus. Each individual word or individual symbol in the textual content is a token, and an entity is the smallest semantic unit consisting of a single token or a number of related tokens. In the process of manually analyzing these bug report samples, we explored some characteristics of software bug-specific named entities:

(1) **The title and description of a bug report are usually made up of these information, i.e., "when", "where", and "what"**. In Table 1, we use three bug reports to illustrate our NER task.
The raw data in Table 1 are the titles of typical bug reports. The second example is the title of bug 1,494,924 shown in Fig. 1. There are more than six entities in just one title, where **the distribution of the entities is very dense**. "Serious" and "Bad" indicate the severity of the bug report; "Junk characters" and "loses JavaDocs" represent the "what" information that indicates the specific symptom of bugs; "on the Restart dialog" and "on google docs" provide the "where" information; "at the end of the installation on RUS OS" and "Java to C++ translation" is the "when" information that indicates the occurrence scenario of the bug. In general, the task of NER is to identify seven types of named entities (i.e., person name, organization, location, time, date, currency, and percentage). These entities are generally nouns. But we notice that **software bug-specific entities contain various parts of speech (POS)**. Most entities are adjectives, adverbs, or nominal forms of verbs except nouns. Therefore, we set two categories for the bug entities: *common adjective* and *common verb*.

(2) Most NER systems process high-quality text that is structured, formal, well-written, grammatically structured, and has few spelling errors. However, bug data is composed of a mixed language environment studded with code, abbreviations, software-specific vocabulary and vague language, which makes NER difficult to handle it. In addition, different users or testers experiencing similar issues

Table 1
Three labeled bug data examples.

Type of data	Data
Raw data	Serious Junk characters on the Restart dialog at the end of the installation on RUS OS.
Annotated data	"Serious *JJ* **B-CA**" "Junk *NNP* **B-GUI**" "characters *NNS* **I-GUI**" "on *IN* **O**" "the *DT* **O**" "Restart *NNP* **B-GUI**" "dialog *NN* **I-GUI**" "at *IN* **O**" "the *DT***O**" "end *NN* **B-GUI**" "of *IN* **O**" "the *DT***O**" "installation *NN* **B-CV**" "on *IN* **O**" "RUS *NNP***B-PF**" "OS *NNP* **B-PF**"
Raw data	Bad performance on google docs caused by fallback/blob image/svg filters.
Annotated data	"Bad *JJ* **B-CA**" "performance *NN* **B-core**" "on *IN* **O**" "google *NN* **B-FW**" "docs *NNS* **I-FW**" "caused *VBD* **B-CV**" "by *IN* **O**" "fallback *VB* **B-core**" "blob *NN* **B-SD**" "image *NN* **I-SD**" "svg *NN* **B-SD**" "filters *NNS* **I-SD**"
Raw data	[HTML5] Java to C++ translation loses JavaDocs.
Annotated data	"HTML5 *NN* **B-SD**" "Java *NN* **B-LA**" "to *IN* **O**" "C++ *NN* **B-LA**" "translation *NN* **B-CV**" "loses *VBZ* **B-CV**" "JavaDocs *NN* **B-API**"

Table 2
Categories for bug-specific entities.

Entity categories		Anno.tag	Examples
Component	core	core	loop code variable type
	GUI	GUI	bottom Junk characters
	Network	NW	Modem Openflow
	I/O	IO	Monitor LPT
	Driver	Dri	FDC HDC HDD
	File System	FS	exFAT copy-on-write
	Hardware	HD	Base Memory ECP
Specific	Language	LA	C C# Python
	API	API	javax.swing.plaf
	Standard	SD	TCP AJAX JSON
	Platform	PF	AMD64 Android
	Framework	FW	NumPY Microsoft WORD
General	defect test	TEST	Static testing SQA
	common adjective	CA	wrong inconsistent
	common verb	CV	break miss reset
	Mobile	MOB	Outgoing call Touch Panel

Table 3
The distribution of the sampled bug reports.

Software	Fixed BR	corpus BR	Sentence	Token	Entity
Mozilla(M)	104K	1,000	3,091	50,029	5,653
Eclipse(E)	21K	400	1,680	19,587	2,326
Apache(A)	35K	1,000	3,349	48,860	5,256
Kernel(K)	15K	200	872	11,860	1,538
Total	175K	2,600	8,992	130,336	14,773

are likely to describe the same bug report in different ways.

Some users are used to using more concise phrase descriptions rather than sentence descriptions such as "Junk characters". There are some other different phrases to express similar meanings just like garbage characters, character corruption or trashed characters. These commonly used fixed combinations such as "Junk characters" are more suitable to be labeled as one entity rather than just "characters" that would be understood as features or letters. Therefore, in the annotation process of our work, **there are quite a few entities composed of multiple words**.

(3) Nowadays, with the development of mobile network technology, mobile devices have proliferated and become more and more popular. **There are a lot of vocabulary words in the bug data that describe mobile software issues**. So we also set a separate category for the bug entities related to mobile devices, i.e. *mobile* in the general class.

(4) Most bugs are detected during software testing. **There are also many entities related to the test process in the bug report**, such as test metrics, test methods, and so on. These entities, unlike other software-specific entities, are proprietary to the domain of software bugs. We add another separate category for bug entities, i.e. *test* in the general class.

3.2. Bug entity classification

In the software bug repository, there are different kinds of bugs. However, few bug tracking systems provide a classification of software bugs. Recently, there are some kinds of software bug classification methods with various purposes (Ye et al., 2018; Terdchanakul et al., 2017; Nagwani, 2015). By investigating into existing studies, we find that existing bug classification methods cannot be directly applied to the classification of bug-specific named entities. For example, some studies set too many categories during the bug classification process such as IEEE Standard Classification for Anomalies 1044–1993 (Ieee, 1994), and it is necessary to understand the content of bugs in order to accurately classify them. Some work classifies the bugs based on bug attributes such as ODC (Orthogonal Defects Classification) (Chillarege et al., 1992) or buggy code source such as Thung's method (Thung et al., 2012). These methods can lead to ambiguity in determining the appropriate entity category.

Component is usually the location of a bug, it is unique and indicated in the bug report. Meanwhile, in bug tracking systems such as Bugzilla, the bug search menu can be also browsed according to the component. **Thus, in the process of human entity annotation, bug classification can be made according to the component information**. Software is divided into application software and system software, and components in different kinds of software projects are inherently different. For compatibility with both software, we divide the component class into the following seven types referring to the work of Tan et al. (Tan et al., 2014), as shown in Table 2. These seven types of components in our work are explained as follows:

- The Core category refers to bugs related to the implementation of core functionality (e.g., loop and variable);
- The GUI category refers to bugs related to graphical user interfaces (e.g., bottom and Junk characters);
- The Network category refers to bugs related to network environment and network communication (e.g., modem and Openflow);
- The I/O category refers to bugs related to I/O handling (e.g., monitor and LPT);
- The Driver category refers to bugs related to device drivers (e.g., FDC and HDC);
- The File System category refers to bugs related to file systems (e.g., exFAT and copy-on-write);

- The Hardware category refers to bugs related to hardware architecture (e.g., memory and ECP).

Many entities (for example, software-specific names) are used to describe bugs that occur on different components. If we mechanically classify these entities according to component types, an overlapped classification will appear. Therefore, we add two other classification classes to adapt these general bug entities. In this way, **we set three classes to cover all categories of bug entities: *component*, *specific*, and *general***. As mentioned in Section 3.1, we define four categories in the general class: *defect test, common adjective, common verb* and *mobile*, including commonly used verbs, commonly used adjectives, entities related to the test process and mobile devices. We classify specific entities from the perspective of software engineering, which may be not only relevant to bugs, such as languages, API, tools and so on. In the annotation process, annotation is performed according to the priority level from *specific/general* to *component*. We first analyze each entity to determine whether it belongs to the specific class or general class; if not, we classify it according to the component class.

For the specific class of bug entities, we classify it into the following five types referring to the work of Ye et al. (Ye et al., 2016), as shown in Table 2, i.e., Language, API, Standard, Platform and Framework. These five categories of the specific property in our work are explained as follows:

- The Language category refers to different types of programming languages (e.g., C and C#);
- The API category refers to API elements of libraries and frameworks that developers use (e.g., javax.swing.plaf);
- The Standard category refers to data formats (e.g., pdf, JSON), design patterns (e.g., Abstract Factory and Observer), protocols (e.g., HTTP), technology acronyms (e.g., Ajax), and so on;
- The Platform category refers to hardware or software platforms (e.g., AMD64 and Android);
- The Framework category refers to software tools, libraries and frameworks that developers use (e.g., NumPY and Microsoft WORD).

3.3. Attention-based BiLSTM-CRF model

Previous works for NER usually include two steps: the first is to construct orthographic features or training word embedding and the second is to employ machine learning methods, such as CRF, SVM, maximum entropy and so forth. The BNER proposed in our previous work belongs to the traditional CRF machine learning method. In this paper, we propose a deep learning-based bug-specific named entity recognition method DBNER. DBNER improves the neural network model proposed by Lample et al. in 2016 BiLSTM-CRF) (Lample et al., 2016b), introduces the attention mechanism to calculate the weight of each entity in the bug report, and aims to solve the problem of inconsistent entity tags caused by lengthy bug documents. The attention-based BiLSTM-CRF model is an end-to-end solution treating the NER task as a sequence labeling problem. The architecture of attention-based BiLSTM-CRF model is illustrated in Fig. 2, which consists of four parts, word representation, BiLSTM encoder, attention layer and CRF layer. In the following paragraphs, we will introduce the details of the model.

3.3.1. Embedding layer

Word embedding is the commonly used language modeling and feature learning technique in NLP. Word embedding technology maps words to real low-dimensional vectors to avoid the problem of high dimensionality or sparse vectors of traditional word vectors. By word embedding, similar words can be more closely related to distance, which can reflect the correlation between words and words, words and contexts, so as to reflect the dependence of words (Paccanaro and Hinton, 2001). The embedding layer is used as the first layer in a neural network model. Given a sequence of training tokens (i.e., a sentence) $X = < x_1, x_2, \cdots, x_m >$, the purpose of the embedding layer is to map vocab indexes to dense vectors:

$$w_t = W^{emb} v_t$$

where v_t is the one hot representation of the token x_t. $W^{emb} \in R^{d \times |v|}$ is the embedding lookup table, where d is the embedding dimension and $|v|$ is the length of v_t.

3.3.2. BiLSTM layer

LSTM (Long Short-Term Memory) is one of the state-of-the-art Recurrent Neural Network (RNN) (Hochreiter and Schmidhuber, 1997). LSTM introduces a structure called the memory cell to solve the problem that ordinary RNN is difficult to learn long-term dependencies in the data. As shown in Fig. 2, in the BiLSTM

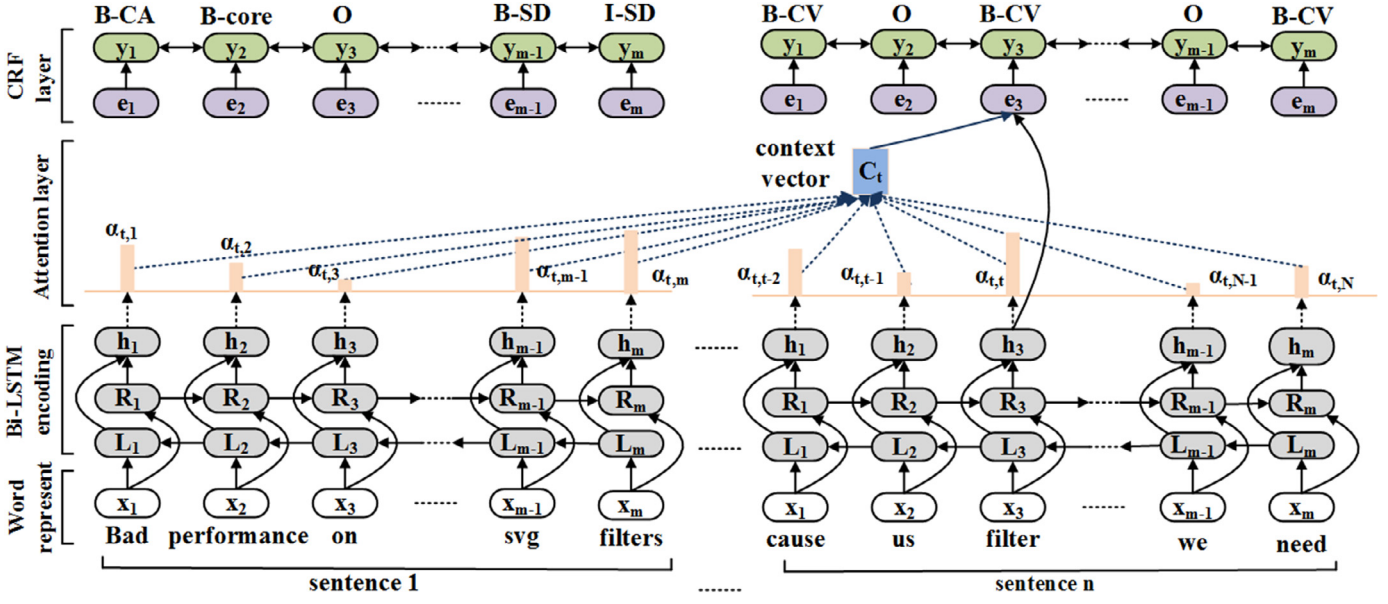

Fig. 2. The architecture of attention-based BiLSTM-CRF model.

encoding layer, a forward LSTM computes a representation r_t of the sequence from left to right at every word vector w_t, r_t represents all the information to the left of word x_t. The other backward LSTM computes a representation l_t of the same sequence in reverse, l_t represents all the information to the right of word x_t. These two distinct networks use same parameters, and then the representation of a word $h_t = [r_t, l_t]$ is obtained by concatenating its left and right context representations. By replacing the word vector w_t with context concatenating vector h_t, the given sentence $X =< W_1, W_2, \cdots, W_m >$ is represented as a sequence of vectors $X =< h_1, h_2, \cdots, h_m >$.

3.3.3. Attention layer

The importance of each token in the sequence is different. The NER object shown in Fig. 2 is the bug 1,494,924 demonstrated in Fig. 1. The bug report consists of three parts: title, description and comment. The title and description sections are succinct and intensive. In contrast, the comment content is scattered, and the entities are sparse. Let's focus on the phrase "svg filters" in title, which is part of the SVG standard and should recognized as an SD category entity. On the one hand, the entity "svg filters", as the cause of the bug, appears simultaneously in different sentences in the title and comment parts of the bug report, and indeed has the greatest impact on the whole bug document. On the other hand, the entity "svg filters" is mentioned in different forms in different sentences, that is, the entity "SVG filter" and entity "SVG" in the comments. If only the context information at the sentence-level is considered, the entity "SVG filter" and the entity "svg filters" are isolated from each other and can be easily identified as entities of different categories. Even more, in Fig. 1, only one comment with rich information is intercepted. The entity "svg filters" appearing in other comments with little information may be easily missed. In order to measure the weight of each token, thereby more accurately identifying entities and reducing the inconsistency of entity tags in lengthy bug documents, we apply an attention mechanism to improve the BiLSTM-CRF model.

Attention mechanism is a recent model that selects the important parts from the input sequence for each target word. We set an attention layer on top of the BiLSTM layer to capture similar word attention at the document-level. The weight generation of each word is guided by a similar attention method proposed by Luo et al. (Luo et al., 2018).

Given a bug document $D =< X_1, X_2, \cdots, X_N >$ consists of n sentences, each sentence is expressed as $X =< x_1, x_2, \cdots, x_m >$ where m is the number of tokens in the sentence. Let N as the total number of tokens, the document is converted to $D =< h_1, h_2, \cdots, h_N >$ after BiLSTM encoding. The attention weight value $\alpha_{t,j}$ is a alignment vector derived by comparing the t^{th} current target token representation h_t with the j^{th} token representation h_j:

$$\alpha_{t,j} = align\big(h_t, h_j\big) = \frac{exp\big(score(h_t, h_j)\big)}{\sum_{i=1}^{N} exp(score(h_t, h_i))}$$

Here, the exp is an exponential function e^x, the $score(h_t, h_j)$ is referred as an alignment function to calculate similarity between the two tokens x_t and x_j:

$$score\big(h_t, h_j\big) = tanh\big(W_\alpha[h_t; h_j]\big)$$

where the weight matrix W_α is a parameter of the model, the $tanh$ is an activation function.

Then, the document-level global vector C_t is computed as a weighted sum of each BiLSTM output h_j:

$$C_t = \sum_{j=1}^{N} \alpha_{t,j} h_j$$

Next, the document-level global vector and the BiLSTM output of the target word are concatenated as a vector $[C_t; h_t]$ to be fed to a $tanh$ function to produce the output of attention layer, W_e is a weight parameter:

$$e_t = tanh(W_e[C_t; h_t])$$

3.3.4. CRF model

We use the final representations e_t as features to make tagging decisions independently, which is extremely effective for each output y_t. However, classification independently is insufficient because the output has strong dependencies, such as our task. When we impose a tag for each token, the sequences of tags contain their own limitation, for example, B-SD indicates that the current token belongs to the *SD* category and is the *Begin* of the entity, I-FW indicates that the current token belongs to the *FW* category and is the *Inside* of the entity. Logically the tag "I-FW" cannot follow behind the tag "B-SD" because "FW" and "SD" belong to different categories, that is, an entity cannot be labeled as two categories. So the DBNER task fails to model by the independent decisions.

The conditional random fields (CRF) proposed by Lafferty (Lafferty et al., 2001) is a good solution for our bug-specific named entity recognition since CRF is one of the most reliable sequence labeling methods, which has shown good performances on different kinds of NER tasks (Ratinov and Roth, 2009; Liu et al., 2011; Wang, 2009). CRF has a strong reasoning ability. In our approach, we use the linear chain CRF model, which makes a first-order Markov independence assumption, and thus can be understood as conditionally-trained finite state machines (FSM)(McCallum and Li, 2003).

Let $X =< e_1, e_2, \cdots, e_n >$ represent a generic input sequence, where e_i is the vector of the i^{th} word; let $Y =< y_1, y_2, \cdots, y_n >$ be a set of FSM states, each of which is associated with a corresponding label, an annotation tag such as the bold annotated data in Table 1. Y denotes all possible tag sequences for a sentence X that can be calculated by the following equation:

$$p(y|X; W, b) = \frac{\prod_{i=1}^{n} \psi_i(y_{i-1}, y_i, X)}{\sum_{y' \in Y} \prod_{i-1}^{n} \psi_i\big(y'_{i-1}, y'_i, X\big)}$$

$$\psi_i\big(y'_{i-1}, y'_i, X\big) = exp\Big(W^T_{y',y} X^i + b_{y',y}\Big)$$

where $W^T_{y',y}$ and $b_{y',y}$ are the weight vector or matrix and bias for the label pair (y', y). We use the maximum conditional likelihood estimation for CRF training. The logarithm of likelihood is given by:

$$L(W, b) = \sum_i logp(y|X; W, b)$$

Maximum conditional likelihood logarithm tries to learn parameters that maximize the log-likelihood $L(W, b)$. In the decoding phrase, we predict the output sequence that obtains the maximum score given for label y^* by the following formula:

$$y^* = argmax_{y' \in Y} p(y|X; W, b)$$

4. Approach

In this section, we present the details of our approach. Fig. 3 shows the framework of the proposed approach*DBNER*. *DBNER* includes three steps. First, we analyze a large number of fixed bug reports and define the classification criterion for the bug-specific named entities as shown in Table 2. In this way, a baseline corpus for bug NER task is constructed. Then, we explore and extract four features from the bug data to optimize the bug NER system. Finally, we employ the attention-based BiLSTM-CRF model to recognize the bug-specific named entities.

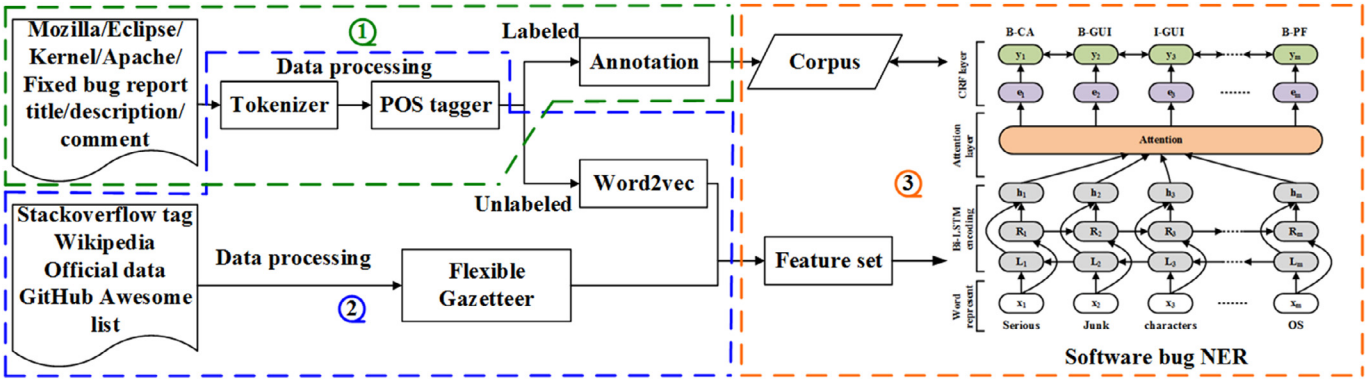

Fig. 3. The general architecture of DBNER.

4.1. Corpus: collection and annotation

A corpus based on bug-specific entity category criteria provides data sets for model training and testing. As shown in Table 3, we collect bug reports with status *fixed* in four projects from Bugzilla by January 2019. Then, through Natural language processing (NLP) techniques, we preprocess these bug text to remove noise data. Finally, we build the baseline bug-specific NER corpus.

4.1.1. Data collection

Because there is no previous research in the bug domain, we must first build a proper corpus for bug NER. We select bug reports from four open source bug repositories: Mozilla, Eclipse, Apache and Kernel, which include application software and system software. In order to ensure the accuracy of bug data, we only collect fixed bug reports. For these bug data, we extract their titles, descriptions and comments. For most of the fixed bug reports, commit messages are always repeated in the comments, so we no longer extract the commit message separately.

There are a lot of entities in the bug document. Subject to manual annotation, we can only sample a portion of bug reports from the collected bug reports to build the corpus. We randomly sample bug reports from various components. Taking into account the diversity and coverage of the bug data, we extract at least one sample for each component. As shown in Table 3, we sample 1000 bug reports from Mozilla, 400 bug reports from Eclipse, 1000 bug reports from Apache and 200 bug reports from Kernel.

4.1.2. Preprocessing

Once we get the collected bug reports, they need to be preprocessed. This process is implemented using the Natural Language Toolkit (NLTK) (Bird, 2006), including removing noise data, tokenization and part-of-speech (POS) tagging. The noise data includes HTML statements, link addresses, information about report submissions, and code snippets displayed in bug reports. For tokenization, each bug report is divided into a series of tokens. We modify the regular expression in the parameter: pattern of the *regexp_tokenize(text, pattern)* (a function in NLTK), to enable it more suitable for the joint structure of the bug-specific entity. For example, “about:memory” is a special page built in Firefox. There are many other similar component names, separated by colons, and should be treated as a complete token. We do not remove the stop words because the removal of stop words would split the semantic links in the context. We reach a size of about 8,992 text sentences consisting of 130,036 tokens as shown in Table 3. For POS tagging, we assign a POS tag to each token of the sampled 2,600 bug reports that build the corpus, as shown in Table 1 in italics. POS tags are then used as the POS feature for the following model learning.

4.1.3. Manual entity annotation

In this process, we use Brat (Stenetorp et al., 2012), a web-based annotation tool for entity annotation. During the annotation process, we use the IOB2 format(Sang, 2002b) with the tags: B (begin), I (inside) and O (outside). As shown in Table 1, “Junk characters” is an entity composed of two tokens, belonging to the *GUI* category. Among them, “Junk” is the first token labeled with *B-GUI*, and “characters” is the second token labeled with *I-GUI*, where *B* and *I* in the tags indicate the *Begin* and *Inside* of the text chunk. Senseless tokens such as “the” outside entities are labeled as “O”. In the corpus, 88% of tokens are labeled as “O” because there are many redundant texts with less information in the comment part.

We only collect fixed bug reports whose root causes can be definitely identified from bug data to ensure correct classification. The entire process is divided into two steps: category definition and manual annotation. We invite eight people to joint the category definition and manual annotation task. These participants are either teachers or graduate students in our lab, and have experience using bug tracking systems and mining bug repositories.

According to the four kinds of bug classification methods discussed in Section 3.2, the participants are divided into four groups to classify the entities with the same data set. When they finish the classification task, they check the results for consistence. If there are disagreements on the classification results, they discuss with each other to determine a final judgment. In this way, the quality of the classification of bug data can be well guaranteed.

After the above process, we start the manual annotation with three participants. We actually go through two rounds. In the first round, we find that certain tokens are difficult to classify, which require some adjustments to the specific definition of each entity category. At the same time, we build a software bug dictionary that references other software data sources, including GitHub, Wikipedia, official websites, and community websites. The bug dictionary is flexibly further divided into gazetteers corresponding to the defined entity categories as shown in Table 2. Software bug entities identified in the annotation process that have been confirmed to belong to the specific class will be continuously updated into the dictionary. Such a constantly enriched dictionary will also feed back the recognition efficiency of bug entities. In the second round, we clean up the annotated data in the first round and annotate them again. We minimize the subjectivity in manual examination through double verification, i.e., each bug report is independently checked at least twice by two participants, and each participant annotate data for four different projects. If the results are inconsistent, they discuss and reach a consensus.

We use Cohen’s Kappa coefficient to measure the level of agreement among participants (Cohen, 1960). Cohen’s Kappa coefficient is widely used as a rating of interrater reliability and it represents the extent to which the corpus constructed in our study correctly represents the bug entity to be recognized. In the annotation, each

bug report has been checked at least twice, and we assign participants according to components, so we no longer calculate the agreement level between participants, but instead calculate the agreement level between the last two check results. The value of the Cohen's Kappa coefficient is computed by the following equation:

$$kappa(K) = \frac{P_O - P_E}{1 - P_E} = \frac{0.918 - 0.414}{1 - 0.414} = 0.86$$

Where $P_O = \frac{119700}{130336} = 0.918$ is the actual agreement rate, which is the sum of the number of entities with consistent recognition results in the 16 entity categories divided by the total number of entities. $P_E = \frac{\sum_{i=1}^{16} a_i \times b_i}{16 \times 16} = 0.414$ is the theoretical agreement rate, where a_i represents the number of entities of each category in the first check result, and b_i represents the number of entities of each category in the second check result. According to the interpretation of kappa coefficient by Manning et al. (Manning et al., 2008), the value of kappa coefficient falling between 0.81 and 1 demonstrates an almost perfect agreement between check results.

Finally, based on the above process, we can build a baseline corpus with categorization and annotation labels for the sampled bug reports. The corpus is further divided into training set and test set for model learning.

4.2. Feature setting

BiLSTM-CRF model can learn features from a dataset automatically during the training process. But considering that the amount of training set is not large enough, simply using tokens and their categories is inadequate for recognizing named entities in the specific context. To facilitate bug NER task, we need to extract domain features not only from the labeled corpus, but also from unlabeled bug reports and ot bug data resources. Our previous approach *BNER* for bug NER (Zhou et al., 2018) introduced five features for the CRF model including POS feature, contextual feature, gazetteer feature, orthographic feature and word embedding feature. All features need to be manually crafted. Compared with *BNER, DBNER* sets four features. On the one hand, *DBNER* uses two sets of BiLSTM model to automatically mine the contextual feature of current token from sentence-level information and the orthographic feature of current token from character-level information. On the other hand, *DBNER* inherits the traditional manual design features, transforming the POS tag of current token into the POS feature, and transforming the matching result between the current token and bug gazetteers into the gazetteer feature. In addition, *DBNER* applies pre-trained word embedding to introduce more external semantic information. Finally, *DBNER* combines the above four features in the form of embedding to represent tokens as inputs to the subsequent encoding layer. The details of the word representation are shown in Fig. 4.

4.2.1. Word-level feature: word embedding

Word embedding can work from a large unlabeled dataset and has shown promising results in lots of NLP tasks (Turian et al., 2010; Socher et al., 2013). In the bug domain, there is no NER corpus. We build a corpus through manual annotation, and the size of the corpus is naturally limited. However, there are huge number of unlabeled bug reports in the bug tracking system. It is more efficient to use pre-trained word embedding than randomly initialized directly.

We use word2vec (Mikolov et al., 2013) which includes two models: CBOW (Continuous Bag-of-Words) model and Skip-gram model. Considering that the Skip-gram model has a better effect on rare words and is suitable for the bug data, we use the Skip-gram model to construct word vectors, calling the gensim library in python[7]. As shown in Table 3, we train word embedding on a large bug dataset consisting of 175,000 fixed bug reports (including bug reports in the corpus), and the parameters are set as follows: size (i.e., word vector dimension) is 100; window (i.e., observed context window size) is 5; workers (i.e., the number of threads working simultaneously) is 4; iter (i.e., total iterations) is 30. We experiment with different settings of word embedding's window size and dimensionality. The results show that embedding with window size 5 usually perform better. We also observe that there are no significant differences between the effectiveness of BiLSTM-CRF model using embedding generated with 300 dimensions as opposed to 100. Then, We employ pre-trained word embedding to initialize our lookup table. These embedding are fine-tuned during the BiLSTM-CRF training.

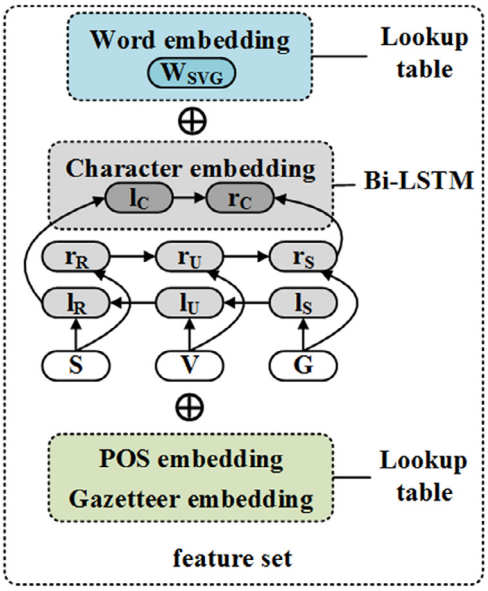

Fig. 4. The combined embedding representation for word "SVG" as input to BiLSTM-CRF model.

4.2.2. Orthographic feature: character embedding

The orthographic feature reflects properties of the current token. Because the colloquial comments and software terminology have a more complex character structure in bug reports, we mine the orthographic feature from the character-level information in addition to the word-level information. Character embedding could alleviate rare word problems and capture helpful morphological information, like prefixes and suffixes.

As shown in Fig. 4, the token "SVG" consists of three upper case letters "S", "V" and "G". We collect all characters forming a character set which contains 26 upper and lower case letters, 10 numbers and 33 punctuation. Unlike the previous traditional methods in which orthographic features are based on hand-engineering, character embedding can be learned while training. First, we initialize a character lookup table randomly from a uniform distribution of $[-0.25, +0.25]$ for the output of 50 dimensions embedding, which drew by a uniform distribution. Then, the character embedding corresponding to every character in a token (i.e., "S", "V" and "G" in the token "SVG") is given in both direct and reverse orders to a BiLSTM layer. At last, the concatenation of the forward and backward representations $w' = [r_c, l_c]$ from the BiLSTM layer is used as the character-level feature of the token.

[7] https://pypi.python.org/pypi/gensim/.

4.2.3. Gazetteer feature

Gazetteers are lists of specific entities in domain. Currently, there are few accepted dictionaries for software and software bugs. Most of the published dictionaries are related to the actual objects in life, such as person, location and organization (Chiticariu et al., 2010). We could not directly use the existing dictionary to identify the software bug-specific named entity, therefore we build a new dictionary consisting of gazetteers for the software bug domain. Mikheev et al. Mikheev et al. (1999) have shown that a small gazetteer list with well known entries is more helpful than a large list listing relatively unknown names, which seldom appear in the text.

We collect rich knowledge from GitHub Awesome Lists[8], Stackoverflow[9], Wikipedia and some software official data to define the gazetteer list. GitHub Awesome Lists are a popular software knowledge source on GitHub developed and maintained by developers. Stackoverflow is a social online community with a significant role in knowledge sharing and acquisition about various software knowledge. In Stackoverflow, each question must be submitted with 3–5 relevant tags by the questioner. So there are a lot of software items in these tags, including some abbreviations and new buzzwords. We build the dictionary list based on the sixteen categories of entities, and then collect software bug domain knowledge based on each gazetteer's list.

Relying on the dictionary, the construction of the gazetteer feature is divided into three steps. First, we perform the longest match between the normalized token sequence and the gazetteers. Then, for each token in the match, the feature is encoded with a BIO (begin, inside, outside) tagging scheme. Finally, we use a lookup table to output a 10-dimensional gazetteer embedding w^g.

4.2.4. POS feature

The POS tag has been obtained during the preprocessing process. Except for the two categories of entities, "*common adjective*" and "*common verb*", the other fourteen categories of bug-specific entities are basically nouns. For example, the POS tag of the word "filter" marked with a blue square at the bottom of Fig. 1 is "VB" (i.e., verb). In Section 3.3.3, we describe the attention mechanism. Through the attention layer, a document-level global vector is constructed to solve the problem of inconsistent entity tags caused by lengthy bug documents. However, this also brings some misjudgments. The two entities "filter" and "svg filters" in Fig. 1 are highly similar. When the entity "svg filters" is recognized as "SD" category, the entity "filter" also tends to be "SD" category. The entities in the corpus should be unique, that is, each entity can belong to only one category. In addition to java filters, the token "filter" also has multiple semantics, such as a parameter in CNN, an operation in wireshark and so on. In this case, it is suitable to perform entity recognition based on the POS tag. On the one hand, when the POS tag of the word "filter" is a noun, it represents a certain kind of software element. We regard the phrase in which the word "filter" is located as an entity, such as "svg filters". Using phrases as entities can effectively assist in identifying the deterministic semantics of the word "filter" in current bug reports. On the other hand, when the POS tag of the word "filter" is a verb, it is no longer necessary to determine its semantics, and the word "filter" can be directly identified as an "CV" entity. we also use a lookup table to output a 25-dimensional POS embedding w^p. Given a sentence $X = < x_1, x_2, \cdots, x_m >$ where m is the length of the sentence, we get the ordinary word representation w'' using the word lookup table. In this way, we obtain a feature-rich word representation by concatenating the four embeddings at position t, i.e., $w_t = [w''_t, w'_t, w^p_t, w^g_t]$.

[8] https://github.com/TheJambo/awesome.

[9] https://stackoverflow.com/tags.

4.3. Attention based BiLSTM-CRF training

In our approach, we apply the tensorflow library and python language to implement our system for attention based BiLSTM-CRF training and learning. For the input bug document to be identified, DBNER represents each word by combining linguistic features (i.e., POS embedding and character embedding) and semantic features from domain resources (i.e., gazetteer embedding and word embedding). Pre-trained embeddings may not always be available for specific words. Words that are out of the embedding vocabulary (OOV) are initialized randomly from a uniform distribution of $[-0.25, +0.25]$ to preserve context relations (Zhang and Wallace, 2017; Jean et al., 2015). We also compare random initialization between $[-0.1, +0.1]$, and the result is not as good as the former $[-0.25, +0.25]$. Through BiLSTM network, we use the context information of sentence sequence to enrich the semantic content of each word embedding. Then, the attention layer on top of the BiLSTM layer calculates the weight and probability distribution of each word from the document level, to prevent the inconsistency of the same word in different sentences. Finally, the weighted word vectors are fed into the CRF layer to decode the best label sequence. The addition of CRF layer can effectively solve the problem of recognizing mostly nested and composite entities in software bug domain.

5. Empirical study

In this section, we present our empirical study to evaluate the effectiveness of our approach for bug-specific named entity recognition. To show the effectiveness of our approach, we propose the following three research questions.

RQ 1: Can our approach effectively recognize bug-specific named entities based on our new built corpus introducing different kinds of features?

In our *DBNER* approach, a deep learning method is applied instead of the previous machine learning method. First of all, we need to analyze whether the deep learning method can really improve the bug NER. Secondly, the advantage of the deep learning method is that it can automatically mine features from text. In our *DBNER* approach, we also introduce additional domain features. We need to analyze whether these domain features are useful for bug NER tasks.

RQ 2: Whether introducing an attention mechanism in our method is effective for the software bug-specific NER?

A bug report is a document consisting of multiple sentences. According to our statistics, each bug report in the corpus contains 4 to 5 sentences. In the preliminary study, we did not consider the weight of the entity, and the problem of inconsistent entity tags appeared in the recognition results. So by introducing an attention mechanism, we focus on the correlation between entities at the document-level, not just the sentence-level, to investigate whether this problem could be solved.

RQ 3: Is our approach effective for recognizing bug entities in projects that there are no available annotated training data?

A generalizable and stable model is crucial for bug NER. In the bug domain, the corpus is a tricky issue. The accurate bug corpus relies on manual annotations. However, new software projects emerge in an endless stream with different functions. It is impossible to build a corpus for each software project. In most cases, when we perform the bug NER task, we have no idea about the background of the software project to which the identification object belongs. So we also need to investigate whether our *DBNER* approach is effective for cross-projects NER task in an unsupervised fashion.

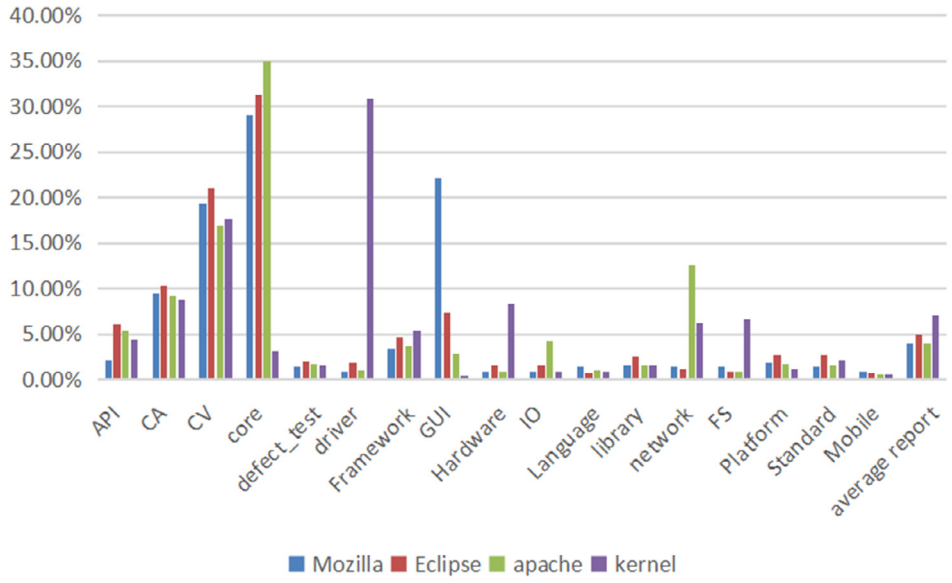

Fig. 5. Proportion of different categories of software bug-specific named entities in the four corpora.

5.1. Data sets

In our study, we extracted bug data from four software projects (i.e., Mozilla, Eclipse, Apache and Kernel). We built a baseline bug-specific corpus by manual annotation as shown in Section 4.1. This corpus is composed of four small corpora belonging to the above four projects, which are not mixed with each other. We sampled different numbers of fixed bug reports from different projects to form the corpus, as shown in Table 3. During the sampling process, we selected bug reports according to the component list. As the products and components in Mozilla and Apache are much more complex than Eclipse and Kernel, therefore, the number of bug reports belonging to the Mozilla and Apache projects is relatively high in the corpus.

The distribution of different categories of software bug-specific named entities in the four corpora is shown in Fig. 5. Although the scale of different corpus is different, the average number of entities per bug report is similar (see the last set of comparisons on the horizontal axis: average report). We compared the top 3 entity categories in each of the four software corpora, which are different from each other. These three projects Mozilla, Eclipse and Apache have a significant number of *core* entities (29% for Mozilla, 31.32% for Eclipse and 35.1% for Apache), and the second is *GUI* in Mozilla (22.08%), *GUI* in Eclipse (7.29%) and *Network* in Apache (12.5%). These three software projects are all application software, but their functionality is different. Eclipse is an open source, extensible development platform. Mozilla includes browsers, mail clients, irc chats and HTML code editors. Apache is one of the mainstream web server software. From this perspective, the distribution of different category of entities is consistent with the functional features of these software projects. The Kernel entity category ratio distribution also confirms this conclusion. Unlike other three software projects, Kernel is the core of an operating system, whose main entities belong to the *Driver* category (30.9%), followed by the *Hardware* category (8.30%).

5.2. Evaluation metrics

We use the standard NER evaluation metrics, i.e., precision, recall and F1, to measure the effectiveness of NER. For each category of named entity, precision (P) measures to what extent the output labels are correct. Recall (R) measures to what extent the named entities in the golden dataset (testing data from the baseline corpus) are labeled correctly. F1-score is the harmonic mean of precision and recall, defined as follows:

$$F1 = \frac{2 \times P \times R}{P + R}$$

5.3. Training set

Because our corpus is relatively small, tratified sampling is used to select 20% of the bug reports based on components to obtain a test set. Then the 5-fold cross-validation method is used for the remaining 80% of the bug reports. We disrupt the remaining bug reports and evenly divided them into five copies, and four of them are selected in turn as the training set and the other as the validation set. The model is validated every 500 mini-batches on the validation set by F1-score for NER. The training time is about one minute per epoch. We run training about 50 epochs and select the best model that has best results on the validation set as the final model after cross-validation. The model is then evaluated on the test set by computing F1-scores and the results will be discussed in Section 5.5.

We chose the mini-batch stochastic gradient descent (SGD), with a fixed learning rate of 0.01 and a gradient norm clipping of 5.0, to optimize our parameters. We also applied other optimizers to compare the results, like Adadelt, Adam and RMSProp. SGD performs best in our model. The parameters are shown as follows:

- The SGD (with minibatch size 100 randomly chosen from training instances) is used to train the parameters.
- The dimension of concatenated embedding is set as 185. Among them, the dimension of word embedding is set to 100, the dimension of character embedding set to 50, the dimension of POS embedding set to 25, and the dimension of gazetteer embedding set to 10.
- DBNER sets the hidden state size of character for LSTM to 25 for forward and backward, and size of word for LSTM to 100 respectively for forward and backward.
- To prevent over-fitting, we use dropout with 0.5.
- Considering the different sentence lengths in the four corpus, we double the maximum average length and set the sentence length to 35 and the word length to 30. We applied truncating or zero-padding to the same length to facilitate batch input network training.

5.4. Experiment design

This paper is an extension of the previous work, five comparative methods including both the traditional machine learning method (i.e., BNER) and deep learning method (i.e., DBNER) were used in the experiment as follows:

CRF model: Our previous work BNER used five features (POS feature, contextual feature, orthographic feature, gazetteer feature and embedding feature) to maximize the accuracy of the CRF model.

BiLSTM-CRF model: This method used only word embedding and character embedding as basic features, without attention mechanism and domain features (i.e., gazetteer embedding and POS embedding).

BiLSTM-CRF model*: This method used all the features, without attention mechanism. The label "*" represents the introduction of domain features.

Att-BiLSTM-CRF model: This method used only word embedding and character embedding as basic features, introducing attention mechanism.

Att-BiLSTM-CRF model*: This method used all the features and attention mechanism.

All methods were conducted on the four corpora (i.e., Mozilla, Eclipse, Apache and Kernel), respectively. Meanwhile, we pre-trained the word embedding on 175,000 sampled fixed bug reports for the five methods.

We answer *RQ1* by measuring the effectiveness of our *DBNER* approach with or without domain features. We answer *RQ2* by measuring the effectiveness of our *DBNER* approach with or without attention mechanism. We answer *RQ3* by performing two cross-project NER experiments on the Att-BiLSTM-CRF* model :

Corpus-specific training: To determine the model generalizability, we trained a model for each corpus and used this to predict the test data in the other corpora. We called the corpus in Mozilla as *M*, the corpus in Eclipse as *E*, the corpus in Kernel as *K* and the corpus in Apache as *A*. First, as there are only 200 bug reports in the corpus *K*, we randomly selected 200 bug reports from each corpus as the testing data. Then, we used the model trained in Mozilla to recognize entities in Eclipse called $M-E$, and used the model trained in Eclipse to recognize entities in Mozilla called $E-M$.

Leave-corpus-out training: To determine the generalizability of the merged-corpora-trained model, we trained a model using all training data of all corpora except one corpus, and tested the model by predicting the left-out corpus test data. For example, we combined *M, E* and *A* into a merged-corpora $M+E+A$. Then, we used the model trained in the merged-corpora $M+E+A$ to recognize entities in the Kernel called $(M+E+A)-K$.

Considering that in most scenarios, the background knowledge of the bug text to be recognized is not available, and the word embedding used in the model in the cross-projects experiment adopts the left-out form, that is, the word embedding is pre-trained on all the unlabeled data except the project to be tested.

5.5. Empirical results

In this section, we show the empirical results to answer the proposed three research questions.

RQ 1: The effect of *DBNER* introducing different features

Table 4 shows the performance of the *DBNER* approach and the previous *BNER* approach trained and tested on the same corpus of each project, respectively. The first column represents the comparative methods. Each cell reports the mean of summary performance measures calculated over multiple runs of 10-fold cross-validation. Comparing the results of CRF model and BiLSTM-CRF model, in the Mozilla, Eclipse and Kernel projects, the BiLSTM-CRF model is not as effective as the CRF model, especially the Eclipse project differs by 2.4%. The deep learning method directly mines features from the data and is limited by the samll size of the bug corpus. In this case, the traditional machine learning method that introduces the word embedding pre-trained on large unlabeled bug data performs better. Comparing the experimental results of the four projects, the size of the Mozilla and Eclipse corpora is larger than the other two projects, and the training results of the four models are relatively better.

Table 4
The results of DBNER for individual-project NER.

Method	Mozilla			Eclipse		
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)
CRF	87.35	89.25	88.29	88.49	85.74	87.09
BiLSTM-CRF	89.19	85.45	87.28	85.14	84.24	84.69
BiLSTM-CRF*	91.49	88.97	90.21	89.21	87.73	88.46
Att-BiLSTM-CRF	90.75	89.64	90.19	90.40	89.52	89.96
Att-BiLSTM-CRF*	91.89	90.31	91.09	91.01	89.67	90.34
Method	Apache			Kernel		
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)
CRF	88.09	85.75	86.90	85.03	79.03	81.92
BiLSTM-CRF	89.62	86.61	88.09	84.74	78.42	81.46
BiLSTM-CRF*	91.57	89.42	90.48	88.73	83.58	86.08
Att-BiLSTM-CRF	90.25	88.93	89.59	86.44	82.75	84.55
Att-BiLSTM-CRF*	92.14	89.60	90.85	89.72	86.79	88.23

Comparing the results of CRF model, BiLSTM-CRF model and BiLSTM-CRF* model, we notice that the addition of domain features can further improve the accuracy of *DBNER* on all of the four projects. Especially in the Kernel project, the F1-score ranges from 81.92% to 81.46%, and then up to 86.08%. Kernel is the core of an operating system, and the largest number of entities in the corpus belong to the Driver and Hardware categories. As mentioned in Section 5.4, we pre-trained the word embedding on 175,000 collected fixed bug reports, and the information of Kernel that can be mined from bug data is relatively small, thus the information of gazetteers can help improve the performance. Similarly, we also performed a comparative experiment on Att-BiLSTM-CRF model and Att-BiLSTM-CRF* model. After introducing domain features, the model gains a small improvement (an improvement of F1-score with 3.6% in Kernel).

In addition, we show the F1-score per entity category to get a more detailed examination of the strengths and weaknesses of our DBNER. As we can see in Fig. 6, the performance of DBNER in different categories of entities are quite different. For example, DBNER for the Driver category performs best in Kernel and worst in the other three projects. In contrast, DBNER for the GUI category performs poor in Kernel and better in other projects. As we can clearly observe from Fig. 5, for the Kernel project, the proportion of entities of Driver category is the highest, and far exceeds that of other projects. At the same time, the proportion of GUI category is low, and lower than other projects. Experiments show that categories of larger proportion are better recognized.

Our entities are divided into three classes: component, specific and general as defined in Section 3.2. There are relatively a few entities of specific type. Therefore, in the testing dataset, if the number of entities of five categories in specific type is fewer than 5, the F1-score of these categories easily gets zero. We also investigated into the performance differences with or without domain features. When only POS embedding is added, there are obvious improvements in general type and component type. That is, POS embedding is helpful in identifying entities of CV and CA categories in the general type, and entities in gerund form in the component type. When only gazetteer embedding is added, there are major improvements in all the three types of entities, especially the re-

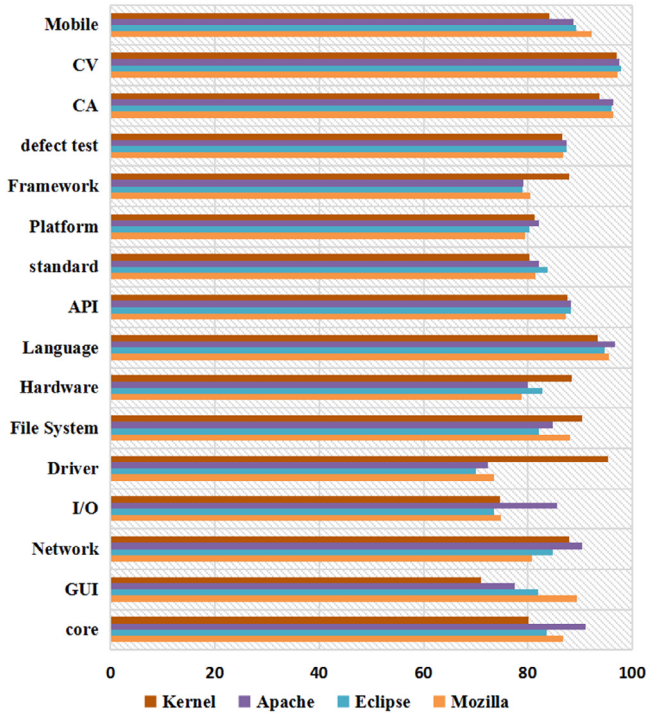

Fig. 6. The performance of DBNER for different categories of entities on four corpora.

call rate of the specific type entities. The three types of entities achieve the best performance when domain features are added.

RQ 2: The effectiveness of *DBNER* based on attention mechanism

We introduce attention mechanism to the BiLSTM-CRF model, and explore the correlation between words from the document-level to solve the problem of inconsistent entity tags. In order to observe whether the NER results are statistically significantly different after the introduction of attention mechanism, we applied the ScottKnott ESD test suggested by Tantithamthavorn (Tantithamthavorn et al., 2017) to disclose the actual improvement and rank difference as shown in Fig. 7. By comparing BiLSTM-CRF model and Att-BiLSTM-CRF model, as well as BiLSTM-CRF* model and Att-BiLSTM-CRF* model, the introduction of attention mechanisms does improve DBNER's effectiveness. Comparing the experimental results of four projects, the Eclipse project gets the most obvious improvement, followed by the Kernel project. Taking the Eclipse project as an example, for the first group of comparison results, the F1-score is increased from 84.69% to 89.96%, and for the second group, the F1-score is increased from 88.46% to 90.34%.

We also compare the BiLSTM-CRF* model and the Att-BiLSTM-CRF model to explore how useful domain features and the attention mechanism are for NER tasks. The BiLSTM-CRF* model performs better in precision and F1 metrics, while the Att-BiLSTM-CRF model has a slightly higher recall rate. We further investigated into the results, randomly sampled 80 bug documents with the inconsistent tag problem from the test results of each project. These 80 bug reports have a total of 526 unique entities, of which 102 entities appear multiple times in the same bug document and have inconsistent labels. Then, we introduced the attention mechanism and reviewed these bug documents. Among them, 54 bug reports eliminated entity inconsistency problem. This result confirms that the attention mechanism is indeed effective.

Table 5
The effect of DBNER for cross-projects NER.

Cross-projects	BNER			DBNER		
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)
M-E	82.04	75.17	78.45	85.26	81.70	83.44
A-E	81.47	74.89	78.04	86.95	82.21	84.51
K-E	77.62	68.37	72.70	76.71	73.25	74.94
(M+A+K)-E	84.63	80.29	82.40	88.42	82.78	85.51
M-A	82.65	76.62	79.52	88.24	81.03	84.48
E-A	80.52	70.59	75.23	83.62	77.42	80.40
K-A	71.03	68.37	69.67	76.70	73.81	75.23
(M+E+K)-A	84.94	78.43	81.56	89.15	84.73	86.88
M-K	74.15	62.38	67.76	76.38	70.83	73.50
E-K	69.90	60.47	64.84	73.86	68.24	70.94
A-K	73.08	62.13	67.16	77.09	72.37	74.66
(M+E+A)-K	75.45	65.58	70.17	80.72	75.91	78.24
E-M	80.13	69.97	74.71	82.03	79.03	80.50
A-M	80.27	74.62	77.34	88.20	80.55	84.20
K-M	70.48	65.70	68.01	78.51	70.92	74.52
(E+A+K)-M	82.79	75.16	78.79	89.59	81.72	85.47

RQ 3: The effectiveness of the cross-project NER

Based on the above experimental results, we select the best performed Att-BiLSTM-CRF* model to study cross-project NER. In cross-project experiments, our *DBNER* model corpus and pre-trained word embedding are from different projects as mentioned above. Although our corpora are composed of sampled bug reports from bug repository, the frequency of entities and words in different corpora changes as the content of the corpus is directly controlled by the functionality and type of the software project. So we investigate whether our approach can effectively recognize the bug-specific named entities based on other projects' corpus. Table 5 shows the results of the two approaches (i.e., CRF model and Att-BiLSTM-CRF* model) for cross-project NER. Taking Apache as an example, the F1-score of the corpus-specific training experiment is 84.48% ($M-A$), 80.40% ($E-A$), and 75.23% ($K-A$). Although the deep learning method DBNER performs better than *BNER* (i.e., 79.52% ($M-A$), 75.23% ($E-A$), and 69.67%($K-A$)), the maximum F1-score (i.e., 90.85% ($A-A$)) for the test data of a specific corpus is only achieved when introducing training data from the same corpus.

Considering the actual situation, the content of software projects is complicated, and many specific information are not public. It is also impossible to spend a lot of effort to annotate a corpus for each project. We show that merging training data from

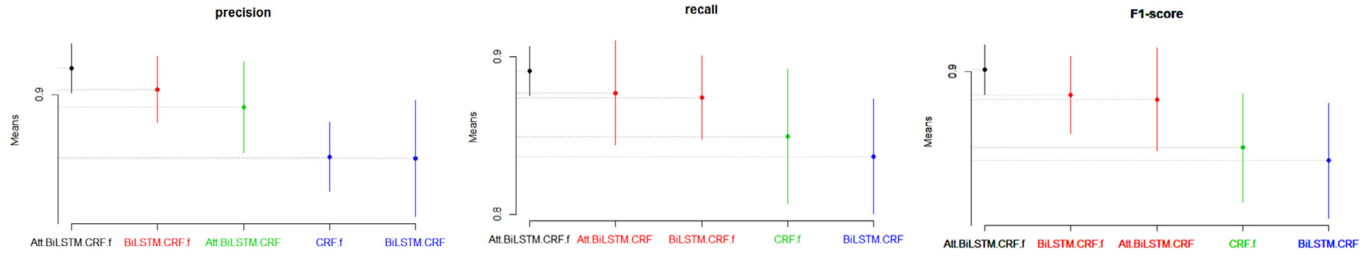

Fig. 7. The ScottKnott ESD test for performance of five NER methods.

multiple available public sources may generate a more generalizable model. As shown in Table 5, in the leave-corpus-out training experiment, the other three corpora in addition to the project to be tested are merged as the training data, and the F1-score is obviously improved, especially for Kernel (an improvement of 4.74% in DBNER). Kernel is the core of an operating system, while the other three software are application software. Many entities in the Kernel corpus are out-of-vocabulary words in the other three project corpora, which cannot be identified in the corpus-specific training experiment, and the recall rate is low (i.e., 62.38% ($M-K$) in BNER, and 72.37% ($A-K$) in DBNER). The merged corpus not only improves the precision but also increases the recall (an improvement of 3.3% in BNER and 3.54% in DBNER).

Based on the results, we can conclude that our *DBNER* approach is also effective for cross-project named entity recognition.

6. Threats to validity

In this section, we discuss possible threats to our study, including construct threats, external threats and internal threats.

- **Construct threats:** In our work, our findings are based on the precision, recall and F1 measures, and other evaluation measures may yield different results. However, these metrics are widely used to evaluate the NER techniques.
- **Internal threats:** We have to build domain-specific corpus due to the limited number of supervised training data. Although the baseline corpus used in this paper are manually identified by different participants, we cannot avoid errors in the process of manual annotations due to subjective factors. In the actual process, although our classification has tried to meet the characteristics of the bug report, there are still some words that cannot be classified accurately.
- **External threats:** We examine the characteristics of bug reports from four open source projects, including Eclipse, Mozilla, Apache and Kernel. All of them employ Bugzilla to store and manage bugs. In Bugzilla, bug reports are submitted by users (contributors). For commercial projects, software companies may use their own bug tracking systems to manipulate bugs. In the commercial bug tracking systems, bug reports may be generated and submitted automatically.

7. Conclusion and future work

Understanding software bug data is important for bug fixing. This paper is an extension of the previous work. In this paper, in order to get finer-granular semantic and syntactic information from bug data, we propose a novel deep learning model for named entity recognition in software bug repository. By investigating into a large number of bug reports, three characteristics of bug-specific named entities are summarized: *various parts of speech (POS), description phrases*, and *solid distribution*. We use the BiLSTM-CRF model as the basic architecture, concatenate word embedding, character embedding, POS embedding and gazetteer embedding to represent words, and make full use of contextual information and bug domain knowledge to improve the accuracy of NER. Since our recognition object is a length bug document composed of multiple sentences, we introduce the attention mechanism to compare the similarity between words from the document level rather than the sentence level to solve the problem of inconsistent entity tags in the same document. The results show that the introduction of domain features and attention mechanism performs better than our previous approach, specifically, the F1-score, recall and precision of *DBNER* for within-project NER have been significantly improved. Moreover, our *DBNER* approach is more suitable for cross-projects NER.

In the future, we will evaluate our approach on more bug reports extracted from other open and commercial software bug repositories. Moreover, we plan to make further improvements in the attention layer and better use domain knowledge. At the same time, we also want to introduce a joint recognition model of entities and relationships to mine more intrinsic information from the bug data. These fine-grained bug knowledge will be used to construct a systematic bug knowledge graph to serve bug repair tasks.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Cheng Zhou: Methodology, Software, Validation, Writing - original draft, Writing - review & editing. **Bin Li:** Conceptualization, Methodology, Supervision, Project administration. **Xiaobing Sun:** Methodology, Supervision, Writing - review & editing.

Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grant No. 61972335, No. 61872312, No. 61472344, No. 61402396 and No. 61611540347, partially by the Natural Science Foundation of Jiangsu Province (BK20150460), partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2018B12, partially by the Jiangsu Qin Lan Project, partially by the Jiangsu “333” Project, partially by the Six Talent Peaks Project in Jiangsu Province under Grant No. RJFW-053, partially by the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project under Grant No. YZU201803, partially by the Open Funds of Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology, under Grant NJ2018014, and partially by the Research Innovation Program for College Graduates of Jiangsu Province (Grant No. KYCX171874).

References

Bird, S., 2006. NLTK: the natural language toolkit. In: ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Sydney, Australia, 17–21 July 2006.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. 3, 993–1022.

Chaparro, O., 2017. Improving bug reporting, duplicate detection, and localization. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume, pp. 421–424.

Chen, H., Lin, Z., Ding, G., Lou, J., Zhang, Y., Karlsson, B., 2019. GRN: gated relation network to enhance convolutional neural network for named entity recognition. CoRR. arXiv: 1907.05611.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M., 1992. Orthogonal defect classification - a concept for in-process measurements. IEEE Trans. Software Eng. 18 (11), 943–956.

Chinchor, N., 1998. Appendix E: MUC-7 named entity task definition (version 3.5). In: Seventh Message Understanding Conference: Proceedings of a Conference Held in Fairfax, Virginia, USA, MUC 1998, April 29 - May 1, 1998.

Chiticariu, L., Krishnamurthy, R., Li, Y., Reiss, F., Vaithyanathan, S., 2010. Domain adaptation of rule-based annotators for named-entity recognition tasks. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP 2010, 9–11 October 2010, MIT Stata Center, Massachusetts, USA, A meeting of SIGDAT, a Special Interest Group of the ACL, pp. 1002–1012.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ Psychol Meas 20 (1), 37–46.

Collobert, R., Weston, J., 2008. A unified architecture for natural language processing: deep neural networks with multitask learning. In: Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5–9, 2008, pp. 160–167.

Dagenais, B., Robillard, M.P., 2012. Recovering traceability links between an API and its learning resources. In: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, pp. 47–57.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2–3, 2010, Proceedings, pp. 31–41.

Hassan, A., Fahmy, H.H., Hassan, H., 2016. Improving named entity translation by exploiting comparable and parallel corpora. Amml.

Hauff, C., Gousios, G., 2015. Matching Github developer profiles to job advertisements. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16–17, 2015, pp. 362–366.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, pp. 392–401.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Huang, Z., Xu, W., Yu, K., 2015. Bidirectional LSTM-CRF models for sequence tagging. CoRR. arXiv: 1508.01991.

Ibrahim, R., Saringat, M.Z., Ibrahim, N., Ismail, N., 2007. An automatic tool for generating test cases from the system's requirements. In: Seventh International Conference on Computer and Information Technology (CIT 2007), October 16–19, 2007, University of Aizu, Fukushima, Japan, pp. 861–866.

Ieee, S., 1994. 1044–1993 - Ieee standard classification for software anomalies. IEEE Standard Indus 9 (2), 1–4.

Jean, S., Cho, K., Memisevic, R., Bengio, Y., 2015. On using very large target vocabulary for neural machine translation. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26–31, 2015, Beijing, China, Volume 1: Long Papers, pp. 1–10.

Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S., Runeson, P., 2016. Automated bug assignment: ensemble-based machine learning in large scale industrial contexts. Empir. Software Eng. 21 (4), 1533–1578.

Lafferty, J.D., McCallum, A., Pereira, F.C.N., 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28, - July 1, 2001, pp. 282–289.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., Dyer, C., 2016. Neural architectures for named entity recognition. CoRR. arXiv: 1603.01360.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., Dyer, C., 2016. Neural architectures for named entity recognition. In: NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12–17, 2016, pp. 260–270.

Li, H., Li, S., Sun, J., Xing, Z., Peng, X., Liu, M., Zhao, X., 2018. Improving API caveats accessibility by mining API caveats knowledge graph. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23–29, 2018, pp. 183–193.

Lin, Z., Xie, B., Zou, Y., Zhao, J., Li, X., Wei, J., Sun, H., Yin, G., 2017. Intelligent development environment and software knowledge graph. J. Comput. Sci. Technol. 32 (2), 242–249.

Liu, X., Zhang, S., Wei, F., Zhou, M., 2011. Recognizing named entities in tweets. In: The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19–24 June, 2011, Portland, Oregon, USA, pp. 359–367.

Luo, L., Yang, Z., Yang, P., Zhang, Y., Wang, L., Lin, H., Wang, J., 2018. An attention-based BILSTM-CRF approach to document-level chemical named entity recognition. Bioinformatics 34 (8), 1381–1388.

Ma, X., Hovy, E.H., 2016. End-to-end sequence labeling via bi-directional LSTM-CNNS-CRF. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7–12, 2016, Berlin, Germany, Volume 1: Long Papers.

Mahalakshmi, G., Vijayan, V., Antony, B., 2018. Named entity recognition for automated test case generation. Int. Arab J. Inf. Technol. 15 (1), 112–120.

Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction to information retrieval. Cambridge University Press.

McCallum, A., Li, W., 2003. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In: Proceedings of the Seventh Conference on Natural Language Learning, CoNLL 2003, Held in cooperation with HLT-NAACL 2003, Edmonton, Canada, May 31, - June 1, 2003, pp. 188–191.

Mikheev, A., Moens, M., Grover, C., 1999. Named entity recognition without gazetteers. In: EACL 1999, 9th Conference of the European Chapter of the Association for Computational Linguistics, June 8–12, 1999, University of Bergen, Bergen, Norway, pp. 1–8.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. CoRR abs/1310.4546.

Nagwani, N.K., 2015. Summarizing large text collection using topic modeling and clustering based on mapreduce framework. J. Big Data 2, 6.

Paccanaro, A., Hinton, G.E., 2001. Learning distributed representations of concepts using linear relational embedding. IEEE Trans. Knowl. Data Eng. 13 (2), 232–244.

Ratinov, L., Roth, D., 2009. Design challenges and misconceptions in named entity recognition. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning, CoNLL 2009, Boulder, Colorado, USA, June 4–5, 2009, pp. 147–155.

Rigby, P.C., Robillard, M.P., 2013. Discovering essential code elements in informal documentation. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, pp. 832–841.

Sang, E.F.T.K., 2002. Introduction to the conll-2002 shared task: language-independent named entity recognition. CoRR cs.CL/0209010.

Sang, E.F.T.K., 2002. Introduction to the conll-2002 shared task: Language-independent named entity recognition. In: Proceedings of the 6th Conference on Natural Language Learning, CoNLL 2002, Held in cooperation with COLING 2002, Taipei, Taiwan, 2002.

Schuh, R.T., Slater, J.A., 1995. True bugs of the world (hemiptera: heteroptera): classification and natural history.. Q. Rev. Biol. (4).

Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18–19, 2013, pp. 2–11.

Socher, R., Bauer, J., Manning, C.D., Ng, A.Y., 2013. Parsing with compositional vector grammars. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4–9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers, pp. 455–465.

Stenetorp, P., Pyysalo, S., Topic, G., Ohta, T., Ananiadou, S., Tsujii, J., 2012. BRAT: a web-based tool for NLP-assisted text annotation. In: EACL 2012, 13th Conference of the European Chapter of the Association for Computational Linguistics, Avignon, France, April 23–27, 2012, pp. 102–107.

Sun, X., Li, B., Leung, H.K.N., Li, B., Li, Y., 2015. MSR4SM: using topic models to effectively mining software repositories for software maintenance tasks. Inf. Software Technol. 66, 1–12.

Sun, X., Liu, X., Li, B., Duan, Y., Yang, H., Hu, J., 2016. Exploring topic models in software engineering data analysis: A survey. In: 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2016, Shanghai, China, May 30, - June 1, 2016, pp. 357–362.

Sun, X., Yang, H., Leung, H., Li, B., Li, H.J., Liao, L., 2018. Effectiveness of exploring historical commits for developer recommendation: an empirical study. Front. Comput. Sci.

Sun, X., Yang, H., Xia, X., Li, B., 2017. Enhancing developer recommendation with supplementary information via mining historical commits. Journal of Systems and Software 134, 355–368.

Sun, X., Zhou, T., Li, G., Hu, J., Yang, H., Li, B., 2017. An empirical study on real bugs for machine learning programs. In: 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4–8, 2017, pp. 348–357.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Empirical Software Engineering 19 (6), 1665–1705.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models (1).

Terdchanakul, P., Hata, H., Phannachitta, P., Matsumoto, K., 2017. Bug or not? bug report classification using n-gram IDF. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17–22, 2017, pp. 534–538.

Thung, F., Lo, D., Jiang, L., 2012. Automatic defect categorization. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15–18, 2012, pp. 205–214.

Turian, J.P., Ratinov, L., Bengio, Y., 2010. Word representations: A simple and general method for semi-supervised learning. In: ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11–16, 2010, Uppsala, Sweden, pp. 384–394.

Wang, Y., 2009. Annotating and recognising named entities in clinical notes. In: ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2–7 August 2009, Singapore, Student Research Workshop, pp. 18–26.

Witte, R., Li, Q., Zhang, Y., Rilling, J., 2007. Ontological text mining of software documents. In: Natural Language Processing and Information Systems, 12th International Conference on Applications of Natural Language to Information Systems, NLDB 2007, Paris, France, June 27–29, 2007, Proceedings, pp. 168–180.

Yan, J., Wang, C., Cheng, W., Gao, M., Zhou, A., 2018. A retrospective of knowledge graphs. Front. Comput. Sci. 12 (1), 55–74.

Yang, H., Sun, X., Li, B., Duan, Y., 2016. Dr_PSF: Enhancing developer recommendation by leveraging personalized source-code files. In: 40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10–14, 2016, pp. 239–244.

Yang, H., Sun, X., Li, B., Hu, J., 2016. Recommending developers with supplementary information for issue request resolution. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016 - Companion Volume, pp. 707–709.

Ye, D., Xing, Z., Foo, C.Y., Ang, Z.Q., Li, J., Kapre, N., 2016. Software-specific named entity recognition in software engineering social content. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016 - Volume 1, pp. 90–101.

Ye, X., Fang, F., Wu, J., Bunescu, R.C., Liu, C., 2018. Bug report classification using LSTM architecture for more accurate software defect locating. In: 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17–20, 2018, pp. 1438–1445.

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

Zhang, Y., Wallace, B.C., 2017. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In: Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27, - December 1, 2017 - Volume 1: Long Papers, pp. 253–263.

Zhou, C., 2018. Intelligent bug fixing with software bug knowledge graph. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04–09, 2018, pp. 944–947.

Zhou, C., Li, B., Sun, X., Guo, H., 2018. Recognizing software bug-specific named entity in software bug repository. In: Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018, pp. 108–119.

Zhou, G., He, T., Zhao, J., Hu, P., 2015. Learning continuous word embedding with metadata for question retrieval in community question answering. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26–31, 2015, Beijing, China, Volume 1: Long Papers, pp. 250–259.

ZHOU Cheng is a student in School of Information Engineering, Yangzhou University. Her current research interests include intelligent bug fixing and software data analysis.

LI Bin is a professor in School of Information Engineering, Yangzhou University. His current research interests include software engineering, artificial intelligence, etc.

SUN Xiaobing is an associate professor in School of Information Engineering, Yangzhou University. His current research interests include intelligent software engineering, software data analytics.