Recognizing Software Bug-Specific Named Entity in Software Bug Repository

Cheng Zhou
School of Information Engineering
Yangzhou University
Yangzhou, China
153362746@qq.com

Bin Li
School of Information Engineering
Yangzhou University
Yangzhou, China
lb@yzu.edu.cn

Xiaobing Sun
School of Information Engineering
Yangzhou University
Yangzhou, China
xbsun@yzu.edu.cn

Hongjing Guo
School of Information Engineering
Yangzhou University
Yangzhou, China
1159107090@qq.com

ABSTRACT

Software bug issues are unavoidable in software development and maintenance. In order to manage bugs effectively, bug tracking systems are developed to help to record, manage and track the bugs of each project. The rich information in the bug repository provides the possibility of establishment of entity-centric knowledge bases to help understand and fix the bugs. However, existing named entity recognition (NER) systems deal with text that is structured, formal, well written, with a good grammatical structure and few spelling errors, which cannot be directly used for bug-specific named entity recognition. For bug data, they are free-form texts, which include a mixed language studded with code, abbreviations and software-specific vocabularies. In this paper, we summarize the characteristics of bug entities, propose a classification method for bug entities, and build a baseline corpus on two open source projects (Mozilla and Eclipse). On this basis, we propose an approach for bug-specific entity recognition called BNER with the Conditional Random Fields (CRF) model and word embedding technique. An empirical study is conducted to evaluate the accuracy of our BNER technique, and the results show that the two designed baseline corpus are suitable for bug-specific named entity recognition, and our *BNER* approach is effective on cross-projects NER.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Named entity recognition, software bug, software bug corpus, CRF model, word embedding

ACM Reference Format:
Cheng Zhou, Bin Li, Xiaobing Sun, and Hongjing Guo. 2018. Recognizing Software Bug-Specific Named Entity in Software Bug Repository. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196321.3196335

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 the bug tracking systems, which are also called bug repository, the whole life cycle of each bug is well tracked from being submitted or opened to being fixed or reviewed[37] .

A lot of work is studied based on mining the bug data from bug reports in the bug repository, such as bug prediction[7][10], bug fix[1][32], bug location[4][29], bug assignment[13, 35][46] and bug categorization[38][28]. These existing studies are used to treat the bug data as textual information and commonly use the information retrival technique, such as vector space model (e.g., TF-IDF[47]), topic model (e.g., LDA[3][34]) or neural network language model (e.g.,word embedding [49]) to search or index the bug information. During these procedures, the bug data is broken into independent words in the text pre-processing steps, and many phrases with associative semantics are broken into individual parts to cause loss of important relationships in the bug data, thus inducing the inaccuracy of the specific software engineering task, such as bug prediction and bug fix.

Hence, existing studies have some limitations to mine the bug data: the overall structural features and internal semantic connections of bug reports are ignored. As a result, this may affect the analysis results when mining the bug repository. For example, for the developer recommendation activity that to identify a suitable developer for fixing a bug, due to the ambiguity of natural language, the same word has different meanings for different developers. Moreover, connections between entities of different bug reports have also been missed. In addition, the new bug report is usually not

[1]https://www.bugzilla.org/

a complete bug report, just some incomplete or inaccurate textual fragments. On the one hand, some useful bug reports that meet the requirements without the keywords that occur in the new bug report may be excluded. On the other hand, some bug reports that contain a few irrelevant keywords in the new bug report but have little to do with the recommended ones are included. In this way, the recommended developer may not fit for the new bug report.

Knowledge graph, used as a new way of knowledge representation and management, which is effective to provide more comprehensive information or knowledge for users[24][42]. Named entity recognition (NER) is a subtask of information extraction used to construct knowledge graph, that seeks to locate and classify named entities in the target text into pre-defined categories [17]. Identifying bug-specific NER can effectively help fill the gap for the bug data in bug repository, i.e., if there is a bug-specific NER for the bug data, the overall structural features and internal semantic connections of bug reports can be recovered, which can effectively improve the effectiveness for mining the software bug repository. However, there is still few studies on software bug-specific NER, the NER task faces the following challenges:

- There is no proper classification method for bug-specific entity and few standard corpus in software bug domain. A more accurate annotated and categorized 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 codes, abbreviations, software specific vocabularies and vague language, etc., which makes the NER more difficult.

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: *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 two open source projects, Mozilla[2] and Eclipse[3]. Then, the bug-specific entity recognition (*BNER*) can be performed based on the baseline corpus. We propose a semi-supervised NER method based on the CRF model and define a set of features for model training based on the baseline corpus. Moreover, we use the word embedding technique to extract features from the whole software bug repository. Finally, we conduct an empirical study on the two projects, Mozilla and Eclipse, to evaluate our *BNER* system. The results show that the designed baseline corpus is useful, and using unsupervised word embeddings as the bug-specific entity's feature has the greatest impact on BNER. In addition, our approach can be effective for cross-project bug-specific named entity recognition. The major contributions of our work are shown as follows:

[2]https://bugzilla.mozilla.org/describecomponents.cgi
[3]https://bugs.eclipse.org/bugs/

- We design a category method for classifying the bug entities into sixteen categories, and develop the baseline bug corpus on such categories to support the bug-specific named entity recognition, which fills the gap of no available bug corpus in this domain.
- To our best knowledge, we are the first to propose a software bug-specific NER approach called *BNER*, which can effectively help identify unknown bug data based on the baseline corpus.
- We evaluate our approach on two open source projects, and the results demonstrate that the designed baseline corpus is suitable for bug-specific named entity recognition, and our *BNER* system is effective for both individual projects and cross-projects.

The remainder of this paper is structured as follows: Section 2 introduces the background. In Section 3, we show our approach to implement NER. In Section 4, we present the empirical study and discuss the empirical results. We present the threats to validity in Section 5. We discuss the related work in Section 6. In Section 7, we conclude this paper and discuss the future work.

2 BACKGROUND

In our work, we propose an NER approach for software bugs in bug repository. In this section, we introduce the background concerning bug report, bug classification, the Conditional Random Fields (CRF) and word embedding techniques to support our bug-specific named entity recognition approach.

2.1 Bug data in bug repository

Nowadays, bug tracking systems, such as Bugzilla, 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. For example, up to December 2017, Bugzilla has managed over 734,000 and 2,604,000 bug reports for Eclipse and Mozilla, respectively. When a bug is found, a bug report is initialized by a contributor to describe its details, e.g., its related product, component, severity, version and the description to reproduce this bug. In bug tracking systems, a bug could be a defect, a new feature, an update to documentation, or a refactoring[10].

When contributors submit a bug report to the software bug repository, the bug report may include these parts: title, description, comments, attachment, importance, reporter, assignee (i.e., fixer), and multiple features such as component and product. 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 was affected by the bug; and the product shows which product was influenced by the bug. To ensure correct entity classification, we only study 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[38].

We randomly sample a diverse set of 1400 bug reports on Mozilla and 400 bug reports on Eclipse covering major components of each projects. Then, we extract the title, description, comments, product and component from each bug report 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 identifying these sampled bug reports, we identify some characteristics of the software bug-specific entity:

(1) Title and description of a bug report are usually made up of "when" "where" "what" "how" parts. In Table 1, we illustrate our BNER task using one example bug report.

The raw data in Table 1 is the title of a typical bug report. There are six entities in just one title, the distribution of the entities is very dense. "Serious" shows the "how" information that indicates the severity of the bug report, "Junk characters" represents the "what" information that indicates the specific status of the bug, "on the Restart dialog" gives the "where" information and "at the end of the installation on RUS OS" is the "when" information that indicates the specific location of the bug. In addition, we notice that software bug entities contain multiple parts of speech (POS). In existing software information extraction studies, the extracted entity is generally a noun. However, most entities in the "how" part are adjectives or adverbs, and entities in the other three parts are verbs or noun form of verbs. Therefore, we set two categories for the bug entities: *common adjective* and *common verb*.

(2) Most NER systems deal with high-quality text that is structured, formal, well written, with a good grammatical structure, and few spelling errors. But bug data lack these qualities, and instead, a mixed language studded with codes, abbreviations, software-specific vocabularies and vague language, which makes NER difficult on this kind of data. Moreover, the same bug report sometimes may be described in different ways by different users or developers who encounter the similar issues.

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 can 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 the problems related to mobile features. So we also set a separate category for the bug entities related to mobile devices, i.e. *mobile in the general class*.

(4) A number of bugs are detected in the process of software testing. There are also lots of test related entities in bug reports, such as quality factors. 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*.

2.2 Bug Classification

In the software bug repository, there are different kinds of bugs. However, few software bug management systems provide the classification method of software bugs. In some studies, there are some kinds of software bug classification methods with various purposes[24][21][20]. By investigating into existing studies, we found that existing bug classification methods cannot be directly applied to the classification of bug entities. For example, some studies set too many categories during the bug classification process such as IEEE Standard Classification for Anomalies 1044-1993[12], 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)[5] or bug source such as National military standard GJB-437. These methodologies can cause ambiguity in determining appropriate entity categories.

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 can be divided into different kinds such as application software and operating software, components in different kinds of software are inherently different. In order to be compatible with these different kinds of software, we divide the component class into the following seven categories referring to the work of Tan et al. [38], as shown in Table 2. These seven categories 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 such as software-specific names are used to describe bugs related to different components. If we mechanically classify these entities according to component categories, an overlapped classification will appear. So we add another two perspectives of the categorization, specific or general. In this way, we have three classes to define the category of a bug entity: *component*, *specific* and *general*. As mentioned in Section 2.1, we define four categories in the general class: *defect test*, *common adjective*, *common verb* and *mobile*, including commonly used verbs, adjectives, entities related to the test process and mobile devices. We classify specific entities from the perspective of the software domain, 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

Table 1: A labeled bug data example

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**"

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

priority level from *specific/general* to *component*. We first identify 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 categories referring to the work of Ye et al. [48], 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).

2.3 Conditional Random Fields (CRF)

In natural language text, entity is the basic information element, which usually indicates the main content of the text. Named Entity Recognition (NER) technology is used to identify entities, determine whether a text string represents a named entity, and classify them into the defined categories. NER has been the integral part of many natural language processing(NLP) applications, such as knowledge graph (KG)[45], Q&A systems and machine translation [8]. Solutions for NER can be basically divided into two kinds: rule-based and statistic-based methods. The rule-based method mainly includes the common persons, locations, organizations and other entities into specialized dictionaries as the reference basis for identification. For entities that are not included in the dictionary, many rules are written manually by experts considering combination characteristic of language to identify them[6]. The statistic-based method needs a larger corpus, but do not need experts. The learning of relevant information is analyzed and learned from the artificial annotated corpus through machine learning algorithm.

The conditional random fields (CRF) proposed by Lafferty [14] 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 named NER tasks [15, 25, 43]. 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)[17].

Let $X =< x_1, x_2, \cdots, x_T >$ be some observed input data sequence (a sequence of tokens or words in bug reports such as the raw data in Table 1); let $Y =< y_1, y_2, \cdots, y_T >$ be a set of FSM states, each of which is associated with a label, an annotation tag such as the bold annotated data in Table 1. For example, **B-GUI** indicates that the current token belongs to the *GUI category* and is the *Beginning* of the entity. Based on the Hammersley-Clifford theorem, CRF computes the conditional probability of a state sequence as follows:

$$P_\Lambda(Y|X) = \frac{1}{Z_x} exp(\sum_{t=1}^{T} \sum_{k} \lambda_k f_k(y_{t-1}, y_t, x, t))$$

Where Z_x is a normalization factor over all state sequences; $f_k(y_{t-1}, y_t, x, t)$ is an arbitrary feature function over its arguments; and λ_k is a learned weight for each feature function.

Typically, the features, f_k, are based on the number of hand-crafted atomic observational tests, for example, word is capitalized, word is "javax.swing.plaf", or word appears in wikipedia lists of software-specific names. A large collection of features is formed by making conjunctions of the atomic tests in certain user-defined patterns. In particular, we use CRF++-0.58[4], a popular CRF toolkit that has been widely used for sequential tagging tasks like NER.

2.4 Word embedding

Word embedding is the commonly used language modeling and feature learning technique in NLP. The advantage of word embedding is that similar words can be more closely related to distance, which can reflect the correlation between word and word, word and context, so as to reflect the dependence of words[23]. Word embedding has shown promising results in lots of NLP tasks, such as named entity recognition, sentiment analysis or parsing [30, 30, 40]).

As mentioned in section 2.1, considering three characteristics of entities in bug reports : *various parts of speech(POS)* (verbs, adjectives and nouns), *description phrases* (a phrase to express a sentence) and *solid distribution* (multiple entities in a sequence observations), we use word2vec [19], a word embedding model based on a neural network (NN) to help recognize the entities in the bug corpus. There are two models for word2vec: CBOW (Continuous Bag-of-Words) model and Skip-gram model. CBOW model uses the surrounding words to predict the current word, while Skip-gram model uses the current word to predict the surrounding words. Considering that the Skip-gram model has a better effect on rare words that fit the bug data context, in this paper, we use the Skip-gram model to construct semantic vectors. Given a sequence of training words $X =< x_1, x_2, \cdots, x_T >$, the objective of the Skip-gram model is to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j=0} logp(x_{t+j}|x_t)$$

c is the size of the training context, which can be a function of the word x_t. T is the context window radius. The context window determines which words are considered for computation of the probability.

To implement the Skip-gram model, we use the gensim library in python[5]. We set a large training dataset consisting of fixed bug reports to train the Skip-gram model, 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 experimented with different settings of word embedding's window size and dimensionality. The results show that embeddings with window size 5 usually perform better. We also observed that there are no significant differences between the effectiveness of CRF model using embeddings generated with 300 dimensions as opposed to 100. Many other similar entities can be identified based on the given entity. For example, the ten entities most similar to the entity we search for "browser" are: NSView, firebug, desktops, closing, window, auto-update, tabbed, nsIEProfileMigrator. cpp, compose and kernel32. dll . In these ten entities, nsIEProfileMigrator. cpp and kernel32. dll are code files related to browser functionality; firebug is the web language for browser; closing, auto-update and compose are operations of browser; tabbed, NSView, desktops and window are part of the browser's user interface.

[4]http://crfpp.sourceforge.net/

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

3 APPROACH

In this section, we present the details of our *BNER* system. Figure 1 shows the framework of the proposed *BNER*, which includes three steps: the first is to build a baseline corpus including definition of the bug categories as shown in Table 2 and classification of each token extracted from bug reports as shown in Table 1. Then, we analyze and extract the features from the baseline corpus, other unlabeled bug reports and external data resources such as Stackoverflow and GitHub Awesome Lists. Finally, we employ the linear chain CRF model to recognize the bug entities.

3.1 Corpus: Collection and Annotation

In the bug tracking system, we scratch all the fixed bug reports in these two software bug repositories by September 2017. Then, we process them through word tokenization, POS tagging and human annotation. Finally, we build the baseline bug-specific corpus.

3.1.1 Data collection. There is still no studies in the NER for bug data, so we first need to build a baseline corpus for software bugs. We selected the bug reports from two open source projects: Mozilla and Eclipse. In order to ensure the accuracy of the bug data, we only collected the fixed bug reports. For these bug data, we extracted their titles, descriptions and comments. There are a lot of entities in these textual contents. We randomly and manually analyzed the bug reports from all components. Taking into account the diversity and coverage of the bug data, we did not strictly follow the consistent proportion during the extraction process, ensuring that each component has at least one sample. As shown in Table 3, we had 1400 bug reports from the Mozilla project and 400 bug reports from the Eclipse project to be two individual bug corpus.

Table 3: The distribution of the sampled bug reports

Software	Fixed BR	Sampled BR	Date of sampling data
Mozilla	104K	1400	2017.9
Eclipse	21K	400	2017.9
Total	125K	1800	2017.9

3.1.2 Preprocessing. Once we get the collected bug reports, a pre-processing process is started. This process is implemented using the Natural Language Toolkit (NLTK)[2], including tokenization and POS tagging. For tokenization, each bug report is divided into a series of tokens. We modified 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 component names like this separated by a colon that should be processed as a complete token. We did not remove the stop words because the removal of stop words would split the semantic links in the context. We reached a size of about 3,091 text sentences consisting of 50,029 tokens on Mozilla and about 1,680 text sentences consisting of 19,587 tokens on Eclipse, as shown in Table 4.

For POS tagging, we assigned part-of-speech (POS) tags to each token for the 1800 bug reports as words in italics shown in Table 1. The corpus is a plain text file, in which each row represents a

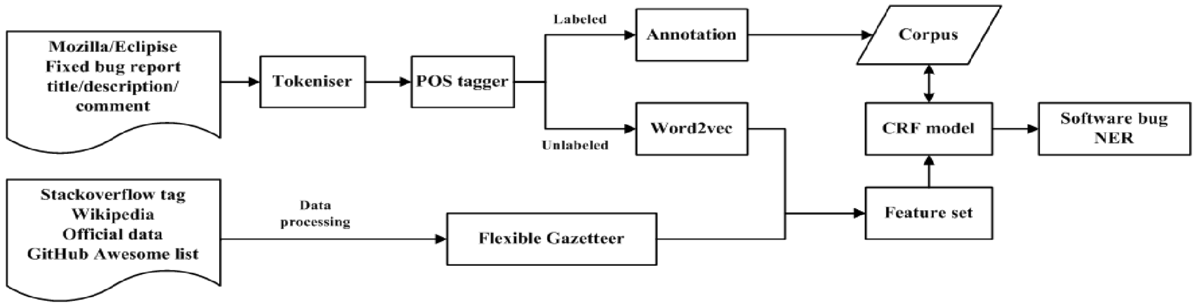

Figure 1: System architecture and pipelines for CRF machine learning-based BNER

token. In Table 1, we use " " to represent a row. POS tags are then used as features for the following CRF learning. In addition to the "*common adjective*" and "*common verb*", other entity categories are mostly nouns, for example, the POS tagger of "serious" belonging to "common adjective" is "JJ" (i.e. Adjective).

Table 4: Data distribution for the bug-specific corpus

Software	Sentence	Token	Entity
Mozilla	3091	50029	5653
Eclipse	1680	19587	2326

After tokenization and POS tagging, we remove some noisy data and extract the POS tags in the bug data.

3.1.3 Human entity annotation. In this process, we used Brat [31], a web-based annotation tool for entity annotation. During the annotation process, we used the IOB2 format[27] 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 word labeled with B-GUI, and "characters" is the second word labeled with I-GUI, where the *B* and *I* in the tags indicate the *Begin* and *Inside* of the text chunk. Other meaningless tokens outside of entities such as "the" are labeled with O. In our corpus, around 85% of the tokens are tagged as outside.

As discussed before, we only collected fixed bug reports whose root causes can be identified from these data sources to ensure correct classification. The entire process is divided into two steps: category definition and manual annotation. We invited eight people to joint the category definition and manual annotation task. These participants are either teachers or graduate students in our lab, and they have some experiences on using bug tracking system and mining the bug repository.

According to the four kinds of bug classification methods discussed in Section 2.2, the participants are divided into four groups to classify the entities with the same dataset. When they finished the classification task, they checked the results for consistence. If there are disagreements on the classification results, they discussed with each other to determine a final judgement. 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 performed this with two rounds. In the first round, we found that some tokens are difficult to categorize, which require some adjustments to the specific definition of each entity category. At the same time, we built a software bug dictionary referring to other data sources such as GitHub, wikipedia and official websites. The gazetteers of the dictionary are classified according to entity categories shown in Table 2. In the second round, we cleaned up the annotated data in the first round and annotated them again. Named entities existed in the gazetteers are directly classified into the same categories. We minimized the subjectivity in manual examination through double verification, i.e., each bug report is examined at least twice by two participants independently and each participant classified both the data of the two different projects. If the results were inconsistent, they discussed and reached a consensus.

Finally, based on the above process, we can build a baseline corpus with categorization and annotation labels for the sampled bugs.

3.2 Feature Extraction

To facilitate named entity recognition, we need to extract the features from the labeled corpus, unlabeled bug reports and external data resources including our new built software bug dictionary. During this process, tokens and POS tags are the basic features to extract. Simply using entities and their types is inadequate for classifying named entities in the specific context. In addition, we extract four other features, including the contextual feature, gazetteer feature, orthographic feature and embedding feature.

3.2.1 Contextual feature. In NER system, contextual feature is the context window referring to the current token, the tokens to its right and to its left in the observation sequence. In our work, we perform the CRF on different window sizes ranging from ±1 to ±5. Based on the cross validation in our empirical study, window size of ±1 performed worse than the remaining four window sizes on all four experimented classifiers. However, we did not find any notable differences among window sizes ranging from ±2 to ±5. Therefore, we selected the shortest and simplest window size in this range, i.e., ±2.

3.2.2 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[6]. Therefore, we do not directly use the existing dictionary to identify the software bug-specific entity, and build a new dictionary for software domain including flexible gazetteers. Mikheev et al. [18] 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 text. We collected rich knowledge from GitHub Awesome Lists [6], Stack Overflow [7], 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 terms in these tags, including some abbreviations and new buzzwords. Referring to GitHub Awesome Lists and wikipedia lists (for example, operating systems, software bugs, computing platform, instruction sets, mobile platform), we build gazetteers including tags on Stack Overflow and some official software and hardware data (API names). Then, the matching results with gazetteers files are used as Gazetteer features.

3.2.3 *Orthographic feature.*

Orthographic feature reflects properties of the current token. There are some special structures for the tokens in bug reports, and we mainly consider the following cases: whether it is initial capitalized or all capitalized, whether it contains a dot, whether it contains a colon and whether it contains digits or just a digital. If a token has a dot, we check if its suffixes match a data format name in the "software standard" gazetteer. If a token has a colon, we match the "component" gazetteer collecting components and products from the software the bug occurs. If it is a digital, we check the "digital" gazetteer by collecting some numbers or combination of numbers and letters that appear in the description of bugs due to data problems, for example, the maximum value "32767" of an Integer.

3.2.4 *Embedding feature.*

Embedding feature contains the information of entities in the unlabeled dataset and model the context by examining surrounding words. We use word embedding as a feature which can capture meaningful syntactic and semantic regularities in NER tasks[41][23][39]. There is a strong association between adjacent words in the textual content of a bug report, where many entities are phrases. In particular, we used *word2vector* to extract word vectors from a large dataset of bug reports. To compare real-valued vectors with categorization labels, we used the k-means algorithm to cluster word vectors into 1024 distinct categories, which is the optimal cluster size suggested by De Vine et al. [41]. We convert all vectors for each token in the training set into a categorization by mapping the token to one of the 1024 categories. If a token was not found in all clusters, we use the category "else" instead [39].

3.3 CRF learning

To recognize the named entities, a widely used technology is to employ the machine learning algorithms. In our work, we perform the BNER based on CRF trained with linguistic features (i.e. tokens, POS tags, contextual and orthographic feature) and semantic features from domain resources (i.e. gazetteer feature and embedding feature) to identify the software bug-specific entities in textual content. Then, we assign each entity to the defined category with annotation tags shown in Table 2 and the *BIO* representation of text chunks.

[6]https://github.com/TheJambo/awesome
[7]https://stackoverflow.com/tags

4 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 two research questions.

RQ 1: Can our approach effectively recognize named entities based on our new built corpus inducing different kinds of features for the CRF model?

To perform the *NER* task, we need to extract features from tokens, the unlabeled large scale bug reports and external software knowledge for the *CRF* model to recognize the software bug-specific entities. In our *BNER* approach, we design the basic feature, contextual feature, Gazetteer feature, orthographic feature, and embedding feature. These features form key elements to identify a bug entity for the baseline bug corpus. So we first evaluate the impact of these different kinds of features for the CRF to recognize the bug-specific named entities.

RQ 2: Is our *BNER* approach effective for software bug-specific NER on cross-projects?

BNER is used for recognizing bug-specific named entities in the bug repository. However, sometimes when we search a bug with BNER, its own bug data may be not available or not enough. At this time, we may need the cross-project bug corpus for BNER. So we also need to investigate whether our *BNER* approach is effective for software bug-specific NER for the cross-project corpus.

4.1 Data Sets

In our study, we selected the bug data from two projects, Mozilla and Eclipse. We first built a small-sized annotated software bug corpus by manual recognition and classification of bug report textual content as shown in Section 3.1. We sampled different numbers of bug reports from different projects to form the corpus, as shown in Table 3. During the sampling process, we selected the bug reports according to the component list. As the products and components in Mozilla are much more complex than Eclipse, we sampled 1400 bug reports from Mozilla and 400 bug reports from Eclipse. As a result, we have 5653 software bug-specific entities in Mozilla and 2326 software bug-specific entities in Eclipse as shown in Table 4 as the baseline corpus.

The distribution of proportion of different categories of software bug-specific named entities in Mozilla corpus and Eclipse corpus is shown in Figure 2. Although the corpus scale is different, the average number of entities per bug report is similar (see the last set of comparisons on the horizontal axis: average report). Based on the generated corpus, we also found that both Mozilla and Eclipse have a significant number of core entities (29% for Mozilla and 31.32% for Eclipse), and the proportion of the GUI in Eclipse (7.29%) is much smaller than that of Mozilla (22.08%). Eclipse is an open source, extensible development platform, while Mozilla includes browsers, mail clients, irc chats, and HTML code editors. From this perspective, the proportion of the GUI category of entities is consistent with the functional features of the two software projects.

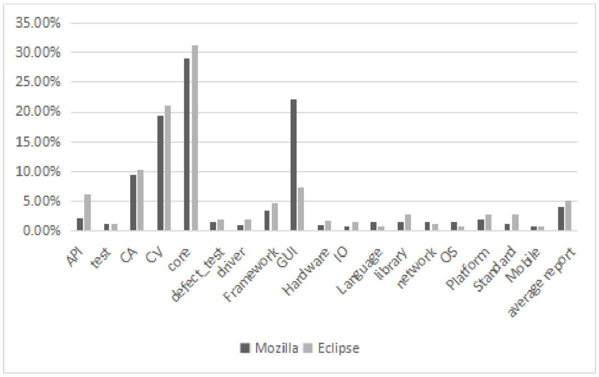

Figure 2: Proportion of different categories of software bug-specific named entities in Mozilla corpus and Eclipse corpus

4.2 Evaluation Metrics

We use the standard NER evaluation metrics, i.e., precision, recall, and F1, to measure the effectiveness of NER. By analyzing the characteristics of a large number of bug reports, two accurate bug-specific corpus covering all components of each project are obtained after manual annotation with cross validation. For each category of named entity, precision (*PRE*) measures to what extent the output labels are correct. Recall (*REC*) measures to what extent the named entities in the golden dataset (testing data from two accurate corpus) are labeled correctly. F1-score is the harmonic mean of precision and recall.

$$F1 = \frac{2 \times PRE \times REC}{PRE + REC}$$

4.3 Experiment Design

We answer *RQ1* by measuring the effectiveness of our *BNER* approach with different kinds of features. The study was conducted on two individual projects, respectively. There are three steps to conduct this study. First, we used the basic features (tokens and POS tags) to train the CRF model. Then, we combined the orthographic feature and gazetteer feature with the basic features. Finally, we induce the embedding feature using word vector and word cluster, respectively. For each step, the corpus was first randomly split into 10 equalized subsets in order to build the training dataset and testing dataset. Then, we used 10-fold cross validation for model training, for which one subset is used as the testing data, and the remaining nine subsets are used as training data. We repeated this process 10 times and produced a single estimation by averaging the ten results. we computed the precision, recall and F1 results, respectively.

We answer *RQ2* by performing the *BNER* task for cross-project NER to show whether the baseline corpus of one project is useful for the other project's bug-specific named entity recognition. We called the unlabeled dataset in Mozilla as *MD* and the unlabeled dataset in Eclipse as *ED*. On the one hand, we randomly selected 100 bug reports from each project 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 $E - M$. This study was also conducted in three steps with different features as discussed above. In addition, we combined *MD* and *ED* into a data set *M&E*, updated embedding feature and retrained the CRF model. Then, we used the model trained in Mozilla to recognize entities in Eclipse and used the model trained in Eclipse to recognize entities in Mozilla again. Considering that other features did not change, we performed this experiment only on the features of word vector or word cluster.

4.4 Empirical results

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

RQ 1: The impact of different features on BNER

Table 5 shows the results of including different features for the bug-specific named entity recognition. From the results, we notice that the addition of each feature can further improve the accuracy of named entity recognition. Specifically, the F1-score of BNER with the basic feature is 68.29%, the orthographic feature 73.03%, the gazetteer feature 80.32%, the word vector feature 86.54% , and the word cluster feature 87.72%. The results further indicate that using the word embedding to learn from the abundant unlabeled data as extra features can improve accuracy of supervised NER model learned from a small amount of corpus. In addition, from the results in Table 5, we can also find that the recall results are improved when the word cluster is used as the feature. The recall results on Mozilla are much higher mainly because the number of bug reports trained for word embeddings in Mozilla is much larger than that of Eclipse.

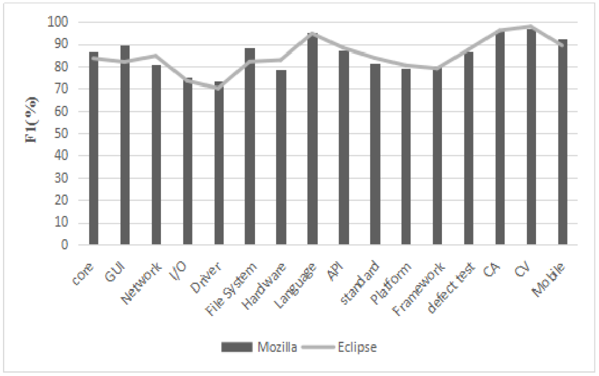

Figure 3: NER results for different categories of entities on Mozilla and Eclipse

In addition, we show the F1-scores per entity category to get a more detailed examination of the strengths and weaknesses of our BNER. Figure 3 shows the results per category. Our entities are divided into three types: component, specific and general as defined in section 2.2. There are relatively a few entities of specific class. Therefore, in the testing data, if the number of entities of

Table 5: Results of different features for independent project NER

Feature	Mozilla			Eclipse			Average
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	F1(%)
Basic	71.34	64.02	67.48	70.04	68.19	69.11	68.29
Orthographic	72.84	72.30	72.57	71.69	75.27	73.48	73.03
Gazetteer	82.19	77.49	79.84	83.75	77.67	80.76	80.32
Vector	87.94	85.42	86.66	88.17	84.73	86.42	86.54
Cluster	87.35	89.25	88.29	88.49	85.74	87.15	87.72

five categories in specific class is fewer than 5, the F1-score of these categories easily gets zero. As we can see in Figure 3, the performance of BNER in different categories of entities are quite different.

We also investigated into the performance differences between word vector and word cluster. The results show that the performance differences on specific category are not obvious. However, performance improvement on component category and general category is significant when word cluster embeddings are used. We consider that the introduction of gazetteer feature greatly improves the precision and recall results for entities of specific class (i.e. standard, platform and framework), so the effect of word embedding improvement is not so obvious. For the other two classes (i.e. component and general), which does not collect the corresponding gazetteers, the improvement of word embedding is obvious.

Based on the results, we see that the performance of the BNER is not well when only using the basic feature. In addition, using other features can improve the effectiveness of the BNER. Specifically, orthographic feature can achieve a good performance, especially with entities that are digital; using gazetteer feature also improves the performance for specific class entities; using unsupervised word embeddings as feature has the greatest impact on the system.

RQ 2: The effectiveness of the cross-project BNER

In cross-project experiments, corpus and word embeddings are from different projects. Although the data sources for both projects all come from the bug reports, there are still many words or entities that have a different word frequency in these two projects. Therefore, if the context information of one project is used to predict the entity of another project, there are many new entities or words with low-frequency that are not well classified, which affects the effectiveness of NER. So we investigate whether our approach can effectively recognize the bug-specific named entities based on cross-project baseline corpus. Table 6 shows the results of using the basic bug corpus for cross-project NER. From the results, we notice that the F1-score of M-E is 79.76% based on word vector and 78.47% based on word cluster, the F1-score of E-M is 75.16% based on word vector and 74.71% based on word cluster. This result shows that using word embeddings as features (about 17% improvement in M-E and 14% improvement in E-M) is useful for BNER. Moreover, when we use the synthetic database of both projects for NER, the effectiveness is improved (about 7.5% improvement in M&E-E and about 13% improvement in M&E-M).

Based on the results, we can conclude that our BNER approach is also effective for cross-project named entity recognition, and the word embedding feature is important for BNER.

5 THREATS TO VALIDITY

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

- **Construct threats:** In our work, our findings are based on the precision, recall and F 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 both datasets of Mozilla and Eclipse used in this paper are manually identified by different participants, we cannot avoid errors in the process of human 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. Another threat is that the number of entities in different categories is unbalanced. For a category with smaller samples, the recall and accuracy results can easily be 0, which makes no sense. Meanwhile, the scale of unlabeled dataset of two kinds of software is quite different.
- **External threats:** We examine the characteristics of bug reports from open source projects, including Eclipse and Mozilla. Both of them are application software, which cannot represent the characteristics of other types of software such as operating system software. On the other hand, two projects 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.

6 RELATED WORK

Named entity recognition, *NER*, the problem of identifying the underlying entities from the data, has been studied for many decades in many communities. Traditional NER approaches primarily identified entities in seven categories (names, organization person, location, organization, time, date, currency and percentage). However, in the software domain, NER has not yet been widely investigated. 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 [16]. Ye et al. proposed an S-NER system using CRF model and Brown clustering to recognize software specific entities such as programming languages,

Table 6: BNER results for cross-project NER

cross-project	no embedding			word vector			word cluster		
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)
M-E	65.32	59.28	62.15	82.16	77.49	79.76	82.04	75.17	78.47
E-M	63.70	57.53	60.46	80.54	70.54	75.16	80.13	69.97	74.71
M&E-E	-	-	-	88.28	85.35	86.79	88.17	86.41	87.28
M&E-M	-	-	-	87.39	86.69	87.04	87.52	89.40	88.45

platforms and libraries on StackOverflow posts [48]. Duc et al. presented an ENER API method choosing the best algorithm from four base algorithms (Rule, CRF-SVM, RNN and Wikification) for each entity recognition [22]. In this paper, we propose a BNER system using CRF model and word embedding to recognize software bug-specific entities of bug reports collected from software bug repository.

In addition, there are some related work focusing on software information extraction. Software engineering data exist in various forms including documentation, SCM documents, source code, bug data and mailing list [11, 33]. 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 [26]. Witte et al. mined the software data documents at semantic level and the extracted information is used in automated population of documentation ontology [44]. Hauff et al. utilized the DBPedia Ontology to extract software concepts from GitHub developer profiles [9]. In this paper, we focused on extracting and analyzing the bug data from bug repository. 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 [29]. Zhang et al. investigated the authorship characteristics of contributors in bug repositories and leverage the authorship characteristics to resolve software tasks. Sun et al. proposed an approach to enhance developer recommendation by extracting expertise and developing habits from historical commits[36]. 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 [4]. In our work, we analyzed the bug data to build a baseline bug corpus, which is further used for named entity recognition

7 CONCLUSION AND FUTURE WORK

Understanding software bug data is important for fixing the bugs in bug repository. In this paper, we proposed an semi-supervised learning approach to implement software bug-specific NER in software bug repository. By investigating into a large number of bug reports, three characteristics of the entity in the report are summarized: *various parts of speech (POS)*, *description phrases*, and *solid distribution.* Based on these characteristics, we divide the bug entity into sixteen categories and build a baseline bug corpus. Then, we develop a *BNER* approach using the linear chain CRF model with the following features: POS tags, contextual feature, orthography feature, gazetteer feature and word embedding feature using both word cluster and word vector. Our *BNER* approach is not based on the keywords, instead, it recognizes entities according to the relationship with the context and the semantic information in the bug data. To demonstrate the effectiveness of our approach, we performed an empirical study on the bug repositories of two popular open: Eclipse and Mozilla. The results show that the features for CRF model are useful, and the accuracy and recall results of each category can reach over 70%, some even more than 80%. Moreover, our BNER approach can be also suitable for cross-project bug-specific named entity recognition.

In the future, we will evaluate our approach on more bug reports extracted from other open and commercial software bug repositories, for example, OS software project. Moreover, we plan to develop a more accurate system by comparing different word representation methods. At the same time, we also want to introduce deep learning techniques such as LSTM (Long Short-Term Memory)-CRF model to implement the NER task.

8 ACKNOWLEDGMENTS

The authors thank all the reviewers and editors for their valuable comments and work. This work is supported partially by Natural Science Foundation of China under Grant No. 61472344, No. 61611540347 and No. 61402396, 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. KFKT2016B21, partially by the Jiangsu Qin Lan Project, partially by the Natural Science Foundation of Yangzhou City under Grant No. YZ2017113, and partially by the Research Innovation Program for College Graduates of Jiangsu Province (Grant No.KYCX171874). Bin Li (lb@yzu.edu.cn) and Xiaobing Sun (xbsun@yzu.edu.cn) are the corresponding authors.

REFERENCES

[1] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar T. Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010.* 97–106.

[2] Steven Bird. 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.*

[3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.

[4] Oscar Chaparro. 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.* 421–424.

[5] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. 1992. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Trans. Software Eng.* 18, 11 (1992), 943–956.

[6] Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Frederick Reiss, and Shivakumar Vaithyanathan. 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. 1002–1012.

[7] Marco D'Ambros, Michele Lanza, and Romain Robbes. 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*. 31–41.

[8] Ahmed Hassan, Hassan Haytham Fahmy, and Hany Hassan. 2016. Improving Named Entity Translation by Exploiting Comparable and Parallel Corpora. *Amml* (2016).

[9] Claudia Hauff and Georgios Gousios. 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*. 362–366.

[10] Kim Herzig, Sascha Just, and Andreas Zeller. 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*. 392–401.

[11] Rosziati Ibrahim, Mohd Zainuri Saringat, Noraini Ibrahim, and Noraida Ismail. 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*. 861–866.

[12] Std Ieee. 1994. 1044-1993 - IEEE Standard Classification for Software Anomalies. *IEEE Standard Indus* 9, 2 (1994), 1 – 4.

[13] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578.

[14] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 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*. 282–289.

[15] Xiaohua Liu, Shaodian Zhang, Furu Wei, and Ming Zhou. 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*. 359–367.

[16] Guruvayur Mahalakshmi, Vani Vijayan, and Betina Antony. 2018. Named entity recognition for automated test case generation. *Int. Arab J. Inf. Technol.* 15, 1 (2018), 112–120.

[17] Andrew McCallum and Wei Li. 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*. 188–191.

[18] Andrei Mikheev, Marc Moens, and Claire Grover. 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*. 1–8.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546

[20] N. K. Nagwani. 2015. Summarizing large text collection using topic modeling and clustering based on MapReduce framework. *J. Big Data* 2 (2015), 6.

[21] Neelofar, Muhammad Younus Javed, and Hufsa Mohsin. 2012. An Automated Approach for Software Bug Classification. In *Sixth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2012, Palermo, Italy, July 4-6, 2012*. 414–419.

[22] Tuan Duc Nguyen, Khai Mai, Thai-Hoang Pham, Minh Trung Nguyen, Truc-Vien T. Nguyen, Takashi Eguchi, Ryohei Sasano, and Satoshi Sekine. 2017. Extended Named Entity Recognition API and Its Applications in Language Education. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, System Demonstrations*. 37–42.

[23] Alberto Paccanaro and Geoffrey E. Hinton. 2001. Learning Distributed Representations of Concepts Using Linear Relational Embedding. *IEEE Trans. Knowl. Data Eng.* 13, 2 (2001), 232–244.

[24] Patrick Pantel and Ariel Fuxman. 2011. Jigs and Lures: Associating Web Queries with Structured Entities. 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*. 83–92.

[25] Lev-Arie Ratinov and Dan Roth. 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*. 147–155.

[26] Peter C. Rigby and Martin P. Robillard. 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*. 832–841.

[27] Erik F. Tjong Kim Sang. 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*.

[28] R. T Schuh and J. A Slater. 1995. True bugs of the world (Hemiptera: Heteroptera): classification and natural history. *Quarterly Review of Biology* 4 (1995).

[29] Ramin Shokripour, John Anvik, Zarinah Mohd Kasirun, and Sima Zamani. 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*. 2–11.

[30] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. 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*. 455–465.

[31] Pontus Stenetorp, Sampo Pyysalo, Goran Topic, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 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*. 102–107.

[32] Xiaobing Sun, Bin Li, Yucong Duan, Wei Shi, and Xiangyue Liu. 2016. Mining Software Repositories for Automatic Interface Recommendation. *Scientific Programming* 2016 (2016), 5475964:1–5475964:11.

[33] Xiaobing Sun, Bixin Li, Hareton K. N. Leung, Bin Li, and Yun Li. 2015. MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks. *Information & Software Technology* 66 (2015), 1–12. https://doi.org/10.1016/j.infsof.2015.05.003

[34] Xiaobing Sun, Xiangyue Liu, Bin Li, Yucong Duan, Hui Yang, and Jiajun Hu. 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*. 357–362.

[35] Xiaobing Sun, Hui Yang, Hareton Leung, Bin Li, Hanchao Jerry Li, and Lingzhi Liao. 2018. Effectiveness of exploring historical commits for developer recommendation: an empirical study. *Frontiers of Computer Science* (07 Feb 2018). https://doi.org/10.1007/s11704-016-6023-3

[36] Xiaobing Sun, Hui Yang, Xin Xia, and Bin Li. 2017. Enhancing developer recommendation with supplementary information via mining historical commits. *Journal of Systems and Software* 134 (2017), 355–368.

[37] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 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*. 348–357.

[38] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and ChengXiang Zhai. 2014. Bug characteristics in open source software. *Empirical Software Engineering* 19, 6 (2014), 1665–1705.

[39] Carson Tao, Michele Filannino, and Özlem Uzuner. 2017. Prescription extraction using CRFs and word embeddings. *Journal of Biomedical Informatics* 72 (2017), 60–66.

[40] Joseph P. Turian, Lev-Arie Ratinov, and Yoshua Bengio. 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*. 384–394.

[41] Lance De Vine, Mahnoosh Kholghi, Guido Zuccon, Laurianne Sitbon, and Anthony N. Nguyen. 2015. Analysis of Word Embeddings and Sequence Features for Clinical Information Extraction. In *Proceedings of the Australasian Language Technology Association Workshop, ALTA 2015, Parramatta, Australia, December 8 - 9, 2015*. 21–30.

[42] Lu Wang, Xiaobing Sun, Jingwei Wang, Yucong Duan, and Bin Li. 2017. Construct bug knowledge graph for bug resolution: poster. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 189–191.

[43] Yefeng Wang. 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*. 18–26.

[44] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. 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*. 168–180.

[45] Jihong Yan, Chengyu Wang, Wenliang Cheng, Ming Gao, and Aoying Zhou. 2018. A retrospective of knowledge graphs. *Frontiers of Computer Science* 12, 1 (2018), 55–74.

[46] Hui Yang, Xiaobing Sun, Bin Li, and Yucong Duan. 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*. 239–244.

[47] Hui Yang, Xiaobing Sun, Bin Li, and Jiajun Hu. 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. 707–709.

[48] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre. 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*. 90–101.

[49] Guangyou Zhou, Tingting He, Jun Zhao, and Po Hu. 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*. 250–259.