Deep API Directive Detection

***Abstract*—Developers increasingly rely on Application Programming Interface (API) libraries to facilitate software development. As constraints and restrictions of APIs in API specifications, API directives seriously impact developers on using APIs. Hence, it could be ideal if API directives can be automatically detected. However, resolving the task of detecting API directives needs to tackle two challenges, i.e., the imbalance between directives and non-directives (Class Imbalance Challenge) and multiple morphologies of directives (Multi-Morphologies Challenge). Even though researchers have proposed an approach relying on syntactic patterns to detect API directives, this approach cannot well tackle the two challenges and its results need to be further improved. In this paper, we propose a novel deep learning based approach named DeepDir. To address the Class Imbalance Challenge, DeepDir first over samples API directives in the class imbalanced training set to balance directives and non-directives. Then, it trains a Bidirectional Long Short Term Memory (Bi-LSTM) network to capture the semantic differences between directives and non-directives to tackle the Multi-Morphologies Challenge. Finally, given a new sentence in an API specification, the trained Bi-LSTM network is used to predict whether it is a directive or not. We have evaluated DeepDir over an annotated API directive corpus with more than 85 thousand sentences from three API specifications. The experimental results reveal that over sampling could boost the performance of DeepDir. In addition, DeepDir significantly improves the state-of-the-art approach by up to 22.83% in terms of F-Measure. When conducting the cross-project prediction, DeepDir achieves a F-Measure of up to 58.98%.**

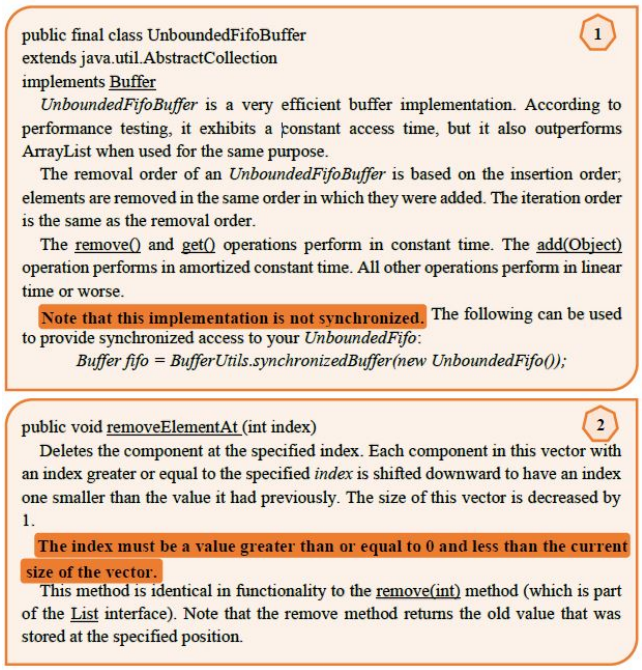


Fig. 1. Two API directive examples

I. Introduction

Software developers tend to reuse existing libraries to facilitate their development process and implement certain functionalities by invoking Application Programming Interfaces (APIs) [1], [2], [3]. However, it remains a challenging task for developers to correctly use APIs, so they often seek for API resources for help [4], [5]. As one of the most important API learning resources, API specifications (also known as API references) detail the instructions on legal API usages with different types of knowledge [1], e.g., functionalities, concepts, and code samples. Out of these knowledge types, developers should particularly pay attention to API directives, i.e., *the natural language statements to describe clear constraints or guidelines that developers should be aware of when programming with APIs* [6]. Once API directives are neglected, fatal development and performance bugs may be easily produced in programming [7].

To better illustrate API directives, we present two examples in Fig. 1. The first example shows the *explanation unit* (i.e., the complete explanation information for a specific API) of the *UnboundedFifoBuffer* class in the commons.collections API specification [8]. The directive in the dark orange background shows that *UnboundedFifoBuffer* is not synchronous. Developers could easily induce a synchronous bug, if this directive is neglected. The second example illustrates the explanation unit of the *Vector.removeElementAt* method in the Java API specification [9]. The directive shows that the range of the argument *index* should be in a specified bound. Otherwise, an *IndexOutOfBoundsException* will be thrown.

From the API directive examples in Fig. 1, we can see that API directives are usually drowned in massive non-directives in API specifications. As to a developer survey investigating the influences of API directives on API usages (see Section II), 85% of developers agree that differentiating directives from the massive non-directives in API specifications is non-trivial. Meanwhile, 94% of developers believe that reminding and highlighting API directives could help them correctly program with APIs and efficiently avoid bugs. Hence, it could be ideal if API directives can be automatically detected. However, resolving the task of detecting API directives in API specifications needs to tackle two challenges as follows.

(1) **Class Imbalance Challenge.** There is an unequal distribution between directives and non-directives, and directives only take up a tiny fraction in API specifications. On the one hand, not all the explanation units contain API directives. On the other hand, most API directives are only described in one or two sentences [6]. For example, there are more than 57 thousand sentences in the Java API specification. However,

only less than 2.8 thousand sentences are manually annotated as directives, which only account for 4.87%.

(2) **Multi-Morphologies Challenge.** Directives utilize different terms and patterns to illustrate the constraints for different APIs [10]. Thus, directives usually have various morphologies. For example, some directives illustrate the constraints of APIs related to *inheritance* and *concurrence*. In contrast, some directives present API constraints related to *arguments* and *return values*.

In a seminal work, Monperrus *et al*. attempt to resolve the task of automatically detecting API directives in API specifications by leveraging a set of syntactic patterns [6]. For example, "invo*" is a proposed syntactic pattern, in which "*" is a wildcard to match zero or more characters. If a sentence in an API specification matches at least one syntactic pattern, it is detected as a directive. Otherwise, it is judged as a non-directive. However, this approach simply treats API specifications as bags-of-words and lacks a deep semantic understanding of API specifications. Due to the Multi-Morphologies Challenge, the syntactic patterns used in an API specification cannot be well adapted to other API specifications. For instance, by applying the syntactic pattern "inherit*", 38 API directives can be detected in the Java API specification. In contrast, no API directive can be discovered by the same syntactic pattern in the commons.collections API specification. Thus, developers have to manually observe a large scale of API specifications to generate and accumulate excessive syntactic patterns. In addition, this approach produces too many false positives, leading to a low Precision value. Hence, there is much room for additional improvement.

The drawbacks of bags-of-words assumption in the above approach motivate us to shift our attention to deep learning models. Deep learning models can obtain a deep understanding of high-level semantics for natural language in two aspects. On the one hand, they can learn the distributed representations of words. Two words with similar semantics are close to each other in the semantic space, so they have a similar effect on deep learning models. On the other hand, deep learning models can distinguish semantic differences between sentences with different word sequences. Hence, sentences with similar word sequence usually achieve similar results in deep learning models. As a result, deep learning models are well suited to resolve the task of API directive detection.

In this paper, we propose DeepDir, a deep learning based approach to automatically detect API directives in API specifications. Specifically, given a class imbalanced training set in which directives only take up a tiny fraction, DeepDir first over samples directives to 50% to balance directives and non-directives, thus tackling the *Class Imbalance Challenge*. Then, DeepDir embeds each word into vectors to learn its distributional representation, which can capture the semantic of the word. Next, DeepDir trains a Bidirectional Long Short-Term Memory (Bi-LSTM) network to fully learn the semantic differences of word sequences in directives and non-directives to tackle the *Multi-Morphologies Challenge*. Finally, given a new sentence in API specification, the trained Bi-LSTM network can predict whether it is a directive or not.

To evaluate the effectiveness of DeepDir, we conduct extensive experiments over a publicly available annotated API directive corpus with more than 85 thousand sentences from three API specifications. The experimental results show that the over-sampling strategy in DeepDir can successfully uncover API directives to achieve better results. DeepDir significantly improves the state-of-the-art approach by up to 25.86% in terms of Precision, by up to 18.03% in terms of Recall, and by up to 22.83% in terms of F-Measure. In addition, when conducting the cross-project prediction, DeepDir can achieve a Precison of up to 65.46%, a Recall of up to 53.70%, and a F-Measure of up to 58.98%.

In summary, this paper makes the following contributions:

- We conduct a developer survey to investigate developers' concerns to API directives. The survey results show that it is non-trivial to detect API directives and developers can really benefit from reminding and highlighting API directives.
- We propose a novel deep learning based approach, i.e., DeepDir, to automatically detect API directives in API specifications. It can balance directives and non-directives, and further capture their semantic differences.
- We conduct extensive experiments to evaluate the effectiveness of DeepDir. Experimental results show that it significantly outperforms the state-of-the-art approach in terms of all the evaluation metrics, i.e., Precision, Recall, and F-Measure.

Outline. The remainder of the paper is structured as follows. First, we present the motivation of this paper in Section II, and show the framework of DeepDir in Section III. Then, we illustrate the experimental setup in Section IV, and elaborate experimental results in Section V. Next, we discuss the threats to validity and related work in Section VI and Section VII, respectively. Finally, we conclude this study and point out future work in Section VIII.

II. Motivation

In this section, we conduct a developer survey to investigate developers' concerns to API directives. Clarifying this question could give a deep insight into how API directives affect API usages in practice.

Survey Design. The survey consists of eight questions, whose details can be found in Table I. The questions in this survey attempt to explore the following three aspects: 1) developers' attitudes towards API specifications; 2) their reactions to API directives; 3) their expectations to API directive detection tools. To improve the response rate and reduce the response time, for each question, we set up some predefined options for developers to choose. The answer options are formed by following a crowd-sourcing method. More specifically, 10 developers are recruited and required to provide their answers to these questions. By collecting and merging the answers, the answer options are generated. In such a way, the answer options are inspired by a group of developers. In addition, an "other" option with a comment field is set up in

TABLE I
SURVEY CONTENT AND RESULTS

Part	Q	Question	Answer Option	Percentage
Part 1	Q1*	How often do you refer to API specifications when facing unfamiliar APIs?	Always	48%
			Often	46%
			Seldom	6%
			Never	0%
Part 2	Q2*	Have you ever paid attention to API directives in API specifications?	Yes	91%
			No	9%
	Q3*	Is it difficult to find API directives in API specifications?	Very Difficult	2%
			Difficult	19%
			Normal	64%
			Easy	13%
			Very easy	2%
	Q4+	What factors prevent you from finding API directives in API specifications?	Too long explanation	52%
			Unstructured text without uniform style	68%
			Useless information embedded	40%
			Lack of focus in a long text	61%
			Other (please specify)	16%
	Q5*	Are automatically reminding and highlighting API directives helpful to avoid bugs?	Very Helpful	46%
			A little helpful	48%
			helpless	6%
Part 3	Q6*	Are API directive detection tools helpful?	Yes	84%
			No	16%
	Q7*	Which is important when detecting API directives?	Higher precision	33%
			Higher recall	18%
			Both higher precision and recall	49%
	Q8*	What suggestions do you have for API directive detection tools?	Blank option	–

*: Exclusive Choice, +: Multiple Choice

some questions to avoid biases to any preconception. Some other surveys also employ the same survey design method, i.e., predefined answers combined with a blank text box (e.g., [11], [12]). If developers cannot find their desired options, they can provide their own unique answers. Hence, the survey is carefully designed, and its results can well reflect the real attitudes of developers to API directives.

Survey Participants. We sample 5,000 Github developers who have ever submitted more than 10 commits to Java projects to participate in the survey. We successfully send the survey invitation by emailing to them with a surveymonkey link [13]. After a period of 15 days, we receive 305 responses in total with a response rate of 6.10%. The response rate is acceptable and considerable with a similar survey investigating API learning obstacles [14]. Additionally, 23 developers reply the email to show their interests and express appreciation to our research. They agree that this survey is helpful to promote the development of the research related to API directives.

Survey Results. The results of the survey are shown in Table I. As to Q1 related to developers' attitudes towards API specifications, most developers frequently consult API specifications when facing unfamiliar APIs, since 48% and 46% of developers choose the "Always" option and the "Often" option, respectively. Hence, API specifications are important resources for developers to learn correct API usages.

As for the questions related to developers' concerns to API directives, 91% of developers have ever paid attention to or dealt with API directives when they program with APIs according to the results of Q2. However, 85% of developers think that finding API directives in API specifications is not easy (see the results of Q3). We can find two main reasons making detecting API directives challenging from the results of Q4, i.e., unstructured text without uniform style (68%) and the lack of focus in a long text (61%) in API specification. Hence, it is meaningful to automatically detect API directives in API specifications. In addition, according to the results of Q5, 94% of developers confirm that automatically reminding and highlighting API directives are helpful to avoid bugs. Therefore, developers can really benefit from detecting API directives in practice.

The third part related to developers' expectations to API directive detection tools consists of three questions. 84% of developers agree that API directive detection tools are helpful (see the results of Q6). As for the performance of API directive detection tools, about a half (49%) of developers confirm that both higher Precision and Recall are important when detecting API directives. In addition, 90 developers provide their suggestions for API directive detection tools. As some of them suggested, "*it would be helpful within an IDE to detect which argument is currently being typed and highlight only text that constrains that argument*" and "*I do think it would be valuable to have some way of writing @throws in a precise mathematical way that could be interpreted by an IDE*". These advices suggest that IDE integration could be convenient for developers. "*provide informative error messages referencing the directives*" suggests that the possible error messages should also be included in API directive detection tools. "*Directives as described in the introduction can usually be described formally*

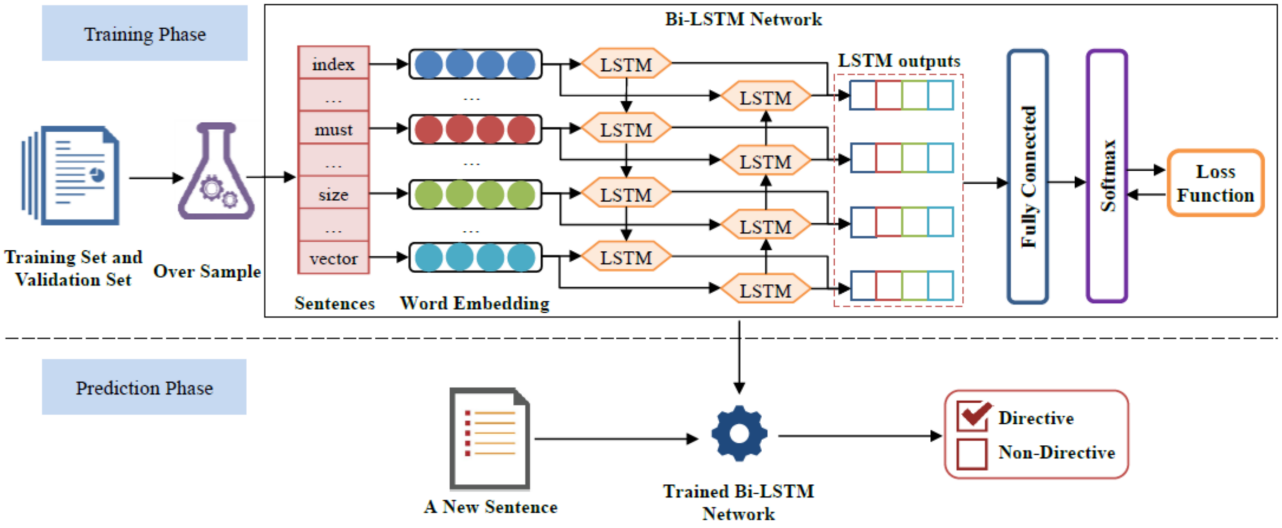


Fig. 2. Workflow of DeepDir

instead of using natural language" recommends that directives should be presented much more formally rather than only using natural language. These constructive suggestions provide new perspectives for API directive related research.

Survey Conclusion. We achieve the following findings from this survey:

1) As important learning resources, API specifications are frequently consulted by developers.
2) API directives are critical for developers to avoid bugs, so developers could benefit from reminding and highlighting API directives.
3) It is not easy for developers to find out API directives in API specifications.

Therefore, it is meaningful to automatically detect API directives in API specifications.

III. Framework

In this section, we describe the technical details of DeepDir.

A. Overview of DeepDir

Fig. 2 shows the workflow of DeepDir, which consists of two phases, i.e., the training phase and the prediction phase. The training phase trains a Bi-LSTM network to learn the semantic differences between directives and non-directives. The prediction phase utilizes the trained Bi-LSTM network to predict whether a new sentence is a directive or not.

The Training Phase. Given a training set and a validation set, DeepDir first over samples directives to make their proportion to be 50% in the training set. Then, for each sentence in the training set, the words in it are represented as vectors using the continuous skip-gram model [15]. Next, the word vectors are sequentially input to the Bi-LSTM network to capture the semantic differences between directives and non-directives. By minimize the loss function in the training set, the Bi-LSTM network can be fully trained. Finally, the trained Bi-LSTM network is evaluated on the validation set to estimate its performance.

The Prediction Phase. Given a new sentence in API specification, the trained Bi-LSTM is used to predict the class label for it, i.e., directive or non-directive.

B. Over Sample

API directives are extremely scarce in API specifications, making it difficult to automatically detect them. According to our statistics, API directives only take up 4.87%, 6.62%, and 11.89% in the Java, JFace, and commons.collections API specifications respectively. To tackle the *Class Imbalance Challenge*, we need first over sample API directives to improve their proportion. In this study, the default over sampling proportion is set to 50%. The reason is that, by setting up the over sampling proportion to 50%, the training set is balanced after over sampling. In such a way, API directives are fully exposed and DeepDir will not be overwhelmed by non-directives. To validate the effectiveness of the default over sampling proportion, we will compare the default value against some other values in Section V.A.

C. The Bi-LSTM Network

1) Architecture: LSTM network is based on the Recurrent Neural Network (RNN), which can learn and model contextual information by using the recurrent connections. RNN can receive arbitrary sequences and propagate the information of previous events. Hence, RNN is suitable for handle sequential data, such as text or speech. RNN utilizes gradient based approaches to train the network. However, with the growth of the number of hidden layers, the backward error is diminished. This phenomenon is called the *vanishing gradient problem*. To resolve this problem, LSTM is proposed with the help of three "gates" in each LSTM unit, including the input gate, the forget gate, and the output gate. The LSTM units are responsible for remembering information over arbitrary time

intervals. The gates in each LSTM unit control the information to be passed among the LSTM units and remember the error without decaying. LSTM has been widely used in the areas of language model, text recognition, and image generation, in which LSTM has achieved considerably good results [16], [17], [18].

API directive detection is inherently dependent on the word sequence of a sentence. Hence, we choose the Bi-LSTM network with word embedding to automatically detect API directives in API specifications. Taking the Bi-LSTM network as a core, the architecture of DeepDir is shown in Fig. 2. First, given a training set and a validation set, each word in a sentence in the training set is embedded as a vector illustrating its distributional representation. In such a way, the sentences in the training set are accordingly embedded into vector sequences, which are regarded as the input layer of the Bi-LSTM network. Then, the vector sequences are input into two Bi-LSTM layers for learning in both the forward and backward directions. The two Bi-LSTM layers exploit the previous and future context regarding the current position, and learn the semantic differences between directives and non-directives from two directions. Next, the outputs of the two Bi-LSTM layers are input into a fully connected layer and a softmax layer. By minimizing the loss function in the training set, the Bi-LSTM network can be fully trained. Finally, the trained Bi-LSTM network is validated on the validation set to evaluate its performance and avoid over-fitting.

2) Word Embedding: The aim of the word embedding layer is to map each word into a equally sized vector in the continuous vector space. Traditional bags-of-words assumption regards the words as independent of each other. However, it cannot reflect the real scenario that a large scale of words are related to each other. Word embedding maps each word into a vector in the vector space, and words sharing similar context are close to each other. Hence, word embedding can measure the semantics of the words to boost the performance of Natural Language Processing (NLP) tasks, e.g., semantic analysis and syntactic parsing [15], [19].

In this study, the words in each sentence are mapped into vectors with 150 dimensions by default, and the value of each dimension is randomly initialized in the range of 0 to 1. The word vectors are trained together with the Bi-LSTM network. This procedure brings two benefits. First. there is no need to manually search for specific text corpora to train the vectors for the words, so it can save a lot of human efforts. Second, the word vectors are trained in the API specification corpus, so it can better reflect the meaning and semantic of each word in API specifications. Therefore, along with the training of the Bi-LSTM, the word vectors can be also learned simultaneously.

3) LSTM Unit: After word vectors are initialized, they are input into the two LSTM layers, which consist of a set of LSTM units. As shown in Fig. 3, each LSTM unit contains four main components, i.e., an input gate, a forget gate, an output gate, and a recurrent connection storing the state of the LSTM unit. Specifically, the input gate controls the new

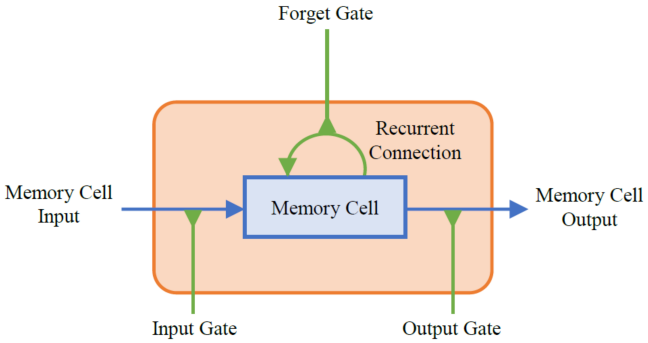


Fig. 3. Diagram of a LSTM unit

information to be stored in the LSTM unit, and the output gate decides the information the LSTM unit needs to output. Meanwhile, the forget gate chooses what information will be discarded from the state of the LSTM unit. The recurrent connection passes the information from the previous to the future. A LSTM layer is regarded as a deep network through time, and two LSTM layers can learn the network from both directions.

4) Dropout: To avoid the over-fitting problem, the dropout technique is often employed to regularize the Bi-LSTM network [20]. Dropout randomly sets some LSTM units to zero during the forward pass of the Bi-LSTM network. Hence, a part of LSTM units do not update in dropout during the training of the Bi-LSTM network. The range of dropout rate is between 0 to 1, where 0 means that no LSTM unit is inactive and 1 means that all the LSTM units are inactive.

5) The Fully Connected Layer and The Softmax Layer: The fully connected layer receives the last sequence output of the two Bi-LSTM layers and obtains scores for the class labels (directive or non-directives). Then, the softmax layer is stacked to turn the scores of class labels into probabilities. Given a real value vector with the size of K, the probability of each value can be calculated by the following softmax function.

$$p(Z_j) = \frac{e^{Z_j}}{\sum_{k=1}^{K} e^{Z_k}} \tag{1}$$

where $p(Z_j)$ means the probability of the score Z_j.

6) The Loss Function: The Bi-LSTM network can be fully trained by minimizing the loss function, and we optimize the loss function by gradient descendant. In this paper, we employ the cross entropy as the loss function, which can be calculated as follows.

$$L = \frac{1}{M} \sum_{i=1}^{M} -(y_i \prime log(y_i) + (1 - y_i \prime) log(1 - y_i)) \tag{2}$$

where M is the number of sentences in the training set. If a sentence is a directive, $y_i \prime = 1$. Otherwise, $y_i \prime = 0$. In addition, y_i is the probability that DeepDir predicts the sentence as a directive.

DeepDir is validated on the validation set by F-Measure in every 100 minibatches, and we select the model that achieves the best results on the validation set as the final model.

D. API Directive Prediction

After the Bi-LSTM network is fully trained, it can be used to predict whether a new sentence in API specification is an API directive or not. In the same way as the training phase, the sentence is first embedded into a word vector sequence. Then, the trained Bi-LSTM network receives the word vector sequence and outputs whether this new sentence is a directive or non-directive.

IV. Experimental Setup

In this section, we first present the hyper parameters in DeepDir, which can help the other researchers reproduce DeepDir. Then, we show the publicly available annotated API directive corpus with its characteristics. Next, we illustrate the details of the baseline approach. Last, we introduce the evaluation method and evaluation metrics employed in this paper.

A. Hyper Parameters in DeepDir

We implement DeepDir using the Tensorflow framework [21]. The hyper parameters in the implementation of DeepDir are shown as follows.

- The size of the word vector (word embedding) is 150.
- The number of LSTM units in each LSTM layer is 256.
- The learning rate for the Bi-LSTM network is 0.001.
- The dropout rate is 0.5.
- The number of epoches is 100 and the batch size is 32.

Similar as [22], [23], we employ these hyper parameter values by default. In the preliminary experiments, we have verified different values of these hyper parameters. With the default hyper parameter values, DeepDir achieves a good result in API directive detection.

B. The Annotated API Directive Corpus

There is a publicly available annotated API directive corpus constructed by Monperrus *et al.*, who empirically study API directives in API specifications [6]. This corpus contains three API specifications, i.e., Java, JFace, and commons.collections. Every API specification consists of a series of HTML web-pages, each of which explains the usage of a specific API. The sentences in these API specifications are manually annotated as directives or non-directives. Monperrus *et al.* construct this corpus by following a standard case-study protocol, which consists of four iterative steps [6]. First, Monperrus *et al.* determine a set of API specifications that have a wide scope to alleviate the bias towards a specific domain. Second, they obtain all the syntactic patterns that are highly likely to describe directives by reading and searching these API specifications. Third, for the syntactic patterns, they manually analyze the matched sentences to decide whether they refer to directives or not. Finally, they formulate the sampling criterion to decide whether a specific syntactic pattern works. By continuously iterating the second to the fourth steps, the final set of syntactic patterns can be decided and the API directives are generated. Following such a protocol, Monperrus *et al.* believe that the quality of the manually constructed annotated corpus can be guaranteed.

TABLE II
Characteristics of the annotated API directive corpus

API Specification	Dir	Non-Dir	Non-Dir/Dir	Length
Java	2,784	54,366	19.53	13.36
JFace	1,250	17,632	14.11	10.23
commons.collections	1,163	8,615	7.41	9.35

The characteristics of the annotated corpus are shown in Table II. *Dir* and *Non-Dir* in the table show the number of directives and the number of non-directives respectively, and *Non-Dir/Dir* shows the ratio between them. Besides, *Length* is the average number of words contained in each sentence. We can see from the table that the three API specifications have different characteristics. The Java API specification is relatively large with more than 57 thousand sentences in total. There is an unequal distribution between directives and non-directives in the Java API specification, i.e., the number of non-directives is 19.53 times as large as that of directives. The JFace API specification is moderate with more than 18 thousand sentences. Still, the number of directives and the number of non-directives in the JFace API specification are imbalanced. The commons.collections API specification is relatively small with 9,778 sentences in total, and the ratio between non-directives and directives is 7.41. By an in-depth analysis on the characteristics of the corpus, we can see that it is a challenging task to detect directives in API specifications, since the number of directives and the number of non-directives are extremely imbalanced.

C. Baseline Approach

In the literature, Monperrus *et al.* attempt to automatically detect API directives in API specifications based on the bags-of-words assumption [6]. They propose an approach to detect API directives by applying a set of 51 manually identified syntactic patterns (i.e., words combined with wildcards). It is the state-of-the-art approach, so we employ it as the baseline approach for comparison. For example, "efficien*" is used as a syntactic pattern to detect API directives, in which "*" stands for a wildcard to match zero or more characters. Hence, this syntactic pattern can match "efficient", "efficiency", and "efficiencies" et al. If a sentence matches this syntactic pattern, it is identified as a directive. Experimental results reveal that this approach can achieve a high Recall value. However, this approach produces too many false positives, so it cannot well balance Precision and Recall.

D. Evaluation Method

In this study, we introduce the widely-used ten-fold cross validation to verify DeepDir, and it is also employed by similar

works [24], [25], [26]. The ten-fold cross validation works as follows. First, we divide each API specification into 10 equally sized folds based on HTML webpages, since it is impractical to put some sentences in a HTML webpage into the training set while the others in the test set simultaneously. Then, similar to [22], [23], the 10 folds are partitioned into three sets with different purposes, i.e. the training set, the validation set, and the test set. Specifically, among the 10 folds, 1 fold is treated as the test set for testing, the fold before the test set is regarded as the validation set (development set) for selecting the best model (if the test set comes from the first fold, then the last fold is regarded as the validation set), and the rest 8 folds are used as the training set for fully training DeepDir. Finally, when each fold is selected as the test set once and this procedure repeats 10 times, the 10 predicted results are averaged to evaluate the performance of DeepDir.

When evaluating the baseline approach, we apply the identified set of syntactic patterns sequentially for each sentence in the corpus, and the baseline approach can decide whether it is a directive or not. After all the sentences in the corpus are checked, we can obtain the results of the baseline approach.

E. Evaluation Metrics

We introduce three widely-used evaluation metrics, i.e., Precision, Recall, and F-Measure, to evaluate the performance of different approaches. Four possible results can be obtained for the sentences in the test set. A sentence can be predicted as a directive when it is actually a directive (True Positive, TP), a sentence is classified as a directive when it is truly a non-directive (False Positive, FP), a sentence is predicted as a non-directive when it is actually a directive (False Negative, FN), and a sentence is classified as a non-directive when it is truly a non-directive (True Negative, TN). Based on the possible outputs, Precision, Recall, and F-Measure can be calculated as follows.

$$Precision = \frac{TP}{TP + FP} \times 100\% \quad (3)$$

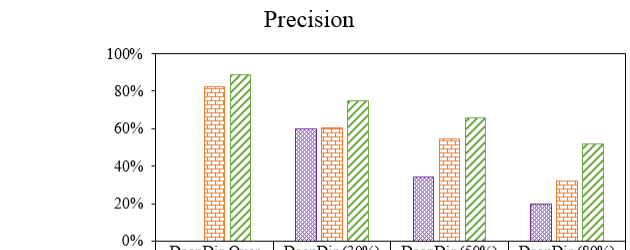
$$Recall = \frac{TP}{TP + FN} \times 100\% \quad (4)$$

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100\% \quad (5)$$

Precision measures how accurate of the approach is in detecting directives. Meanwhile, Recall assesses how complete of the approach is in identifying directives. In addition, F-Measure is a synthetic indicator taking both Precision and Recall into consideration, since there is a tradeoff between Precision and Recall.

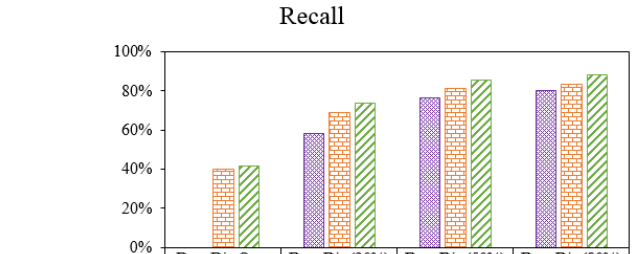
V. Experimental Results

In this section, we investigate three Research Questions (RQs) to explore the performance of DeepDir.



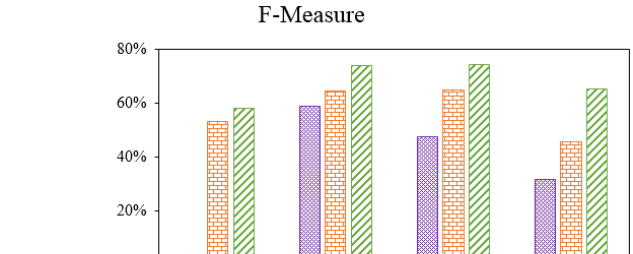
	DeepDir-Over	DeepDir (30%)	DeepDir (50%)	DeepDir (80%)
Java	0%	60.02%	34.43%	19.67%
JFace	82.12%	60.26%	54.61%	32.04%
commons.collections	89.01%	74.88%	65.82%	51.98%

Fig. 4. Precisions of different over sampling proportions



	DeepDir-Over	DeepDir (30%)	DeepDir (50%)	DeepDir (80%)
Java	0%	58.33%	76.17%	80.02%
JFace	39.88%	69.08%	81.09%	83.12%
commons.collections	41.54%	73.64%	85.44%	88.06%

Fig. 5. Recalls of different over sampling proportions



	DeepDir-Over	DeepDir (30%)	DeepDir (50%)	DeepDir (80%)
Java	0%	58.62%	47.39%	31.49%
JFace	53.18%	64.31%	64.90%	45.66%
commons.collections	57.89%	73.97%	74.30%	65.22%

Fig. 6. F-Measures of different over sampling proportions

A. Investigation to RQ1

RQ1: What is the influence of the over sampling proportion on DeepDir?

Motivation. To tackle the *Class Imbalance Challenge*, DeepDir employs an over sampling strategy, which over samples API directives to make their proportion to be 50%. In such a way, the training set is balanced in terms of class label after over sampling. To investigate how the over-sampling proportion affects DeepDir, we set up this RQ.

Approach. We over sample directives to 50% by default. To investigate the influence of the over sampling proportion on DeepDir, we compare the default over sampling proportion against the other two proportions, i.e., 30% and 80%. In

TABLE III
COMPARISON RESULTS BETWEEN DIFFERENT APPROACHES

API Specification	Precision			Recall			F-Measure		
	Baseline	DeepDir	Improvement	Baseline	DeepDir	Improvement	Baseline	DeepDir	Improvement
Java	15.18%	34.43%	+19.25%	73.81%	76.17%	+2.36%	25.18%	47.39%	+22.21%
JFace	28.75%	54.61%	+25.86%	78.40%	81.09%	+2.69%	42.07%	64.90%	+22.83%
commons.collections	49.37%	65.82%	+16.45%	67.41%	85.44%	+18.03%	57.00%	74.30%	+17.30%
Average	31.10%	51.62%	+20.52%	73.21%	80.90%	+7.69%	41.42%	62.20%	+20.78%

addition, we also define a variant of DeepDir, i.e., DeepDir-Over, which removes the over sampling strategy and keeps the other components the same. By comparing DeepDir with DeepDir-Over, the effectiveness of the over sampling strategy can be shown.

Results. Fig. 4, Fig. 5, and Fig. 6 show the detailed results of DeepDir with different over sampling proportions for each API specification. DeepDir (X%) means DeepDir with the over sampling proportion of X%. For example, DeepDir (30%) shows DeepDir with the 30% over sampling proportion. We can see that along with the increase of the over sampling proportion, Precisions of DeepDir show downward trends in the three API specifications. For instance, when inputting the class imbalanced training set, DeepDir-Over achieves a Precision of 89.01% in the commons.collections API specification. When over sampling API directives to 30%, DeepDir (30%) achieves a Precision of 74.88%. Meanwhile, the Precisions of DeepDir (50%) and DeepDir (80%) drop to 65.82% and 51.98% in the same API specification respectively. However, there is an exception, i.e., DeepDir-Over achieves a Precision of 0% in the Java API specification. This is possibly because the proportion of directives in the Java API specification is the smallest (i.e., 4.87%), making DeepDir hard to learn the semantic differences between directives and non-directives. Hence, DeepDir without over sampling tends to predict all the sentences as non-directives in the Java API specification.

In contrast to Precisions, Recalls achieved by DeepDir show upward trends. For example, DeepDir-Over only achieves a Recall of 39.88% in the JFace API specification. When over sampling API directives to 30%, DeepDir (30%) obtains a Recall of 69.08%. In addition, the Recalls of DeepDir (50%) and DeepDir (80%) rise to 81.09% and 83.12% in the JFace API specification respectively.

Some works also observe similar phenomena that Precisions and Recalls show downward trends and upward trends respectively along with the increase of the over sampling proportion [27], [28]. The reason may be that, along with the growth of the over sampling proportion, more API directives are added to the training set. Hence, DeepDir tends to predict more sentences as directives and may achieve more false positives. In such a way, high over sampling proportion increases Recalls and decreases Precisions of DeepDir.

In terms of F-Measure, DeepDir achieves the best results when the over sampling proportion is 50% in two of three API specifications. In this situation, DeepDir is trained by the class balanced training set, so DeepDir can effectively balance Precision and Recall to achieve better F-Measure. For example, DeepDir achieves F-Measures of 64.90% and 74.30% in the JFace and commons.collections API specifications, and the two F-Measure values are the highest in all the over sampling proportions.

From the trends of Precisions, Recalls, and F-Measures of DeepDir along with the growth of the over sampling proportion, we can see that DeepDir is flexible to adapt to different situations. The users of DeepDir can adjust the over sampling proportion according to their conditions. If they want to detect API directives as accurately as possible, they can decrease the over sampling proportion. In contrast, if they want to detect API directives as completely as possible, they may increase the over sampling proportion. Meanwhile, if developers care about both Precision and Recall, they can select a moderate over sampling proportion.

When comparing different API specifications, we can see that the Java API specification is the most difficult one to detect API directives and the commons.collections API specification is the easiest one. For example, DeepDir achieves a F-Measure of 47.39% in the Java API specification. In contrast, in the commons.collections project, the F-Measure can rise up to 74.30%. We can find the underlying reason from Table II. The Java API specification has the most class imbalanced data on average, i.e., the ratio of non-directives and directives is 19.53. In contrast, in the commons.collections API specification, the ratio is only 7.41 on average.

Conclusion. The over sampling strategy is effective in DeepDir. DeepDir can adapt to different situations by adjusting the over sampling proportion.

B. Investigation to RQ2

RQ2: How much improvement can DeepDir achieve against the baseline approach?

Motivation. As we described, there is a baseline (state-of-the-art) approach to automatically detect API directives in API specifications. To investigate whether DeepDir is superior to the baseline approach and how much the improvement is, we set up this RQ.

Approach. We follow the procedures of the baseline approach to implement it. The authors of the baseline approach report the set of syntactic patterns [6], so we can apply them to the sentences in API specifications to detect API directives. If a sentence matches at least one syntactic pattern, it is detected as a directive. Otherwise, it is regarded as a non-directive.

Results. Table III shows the detailed comparison results between the baseline approach and DeepDir. We also present the improvement of DeepDir against the baseline approach in terms of the three evaluation metrics. From the table we can see that, DeepDir is superior to the baseline approach in terms of all the metrics. For example, in the commons.collections API specification, the baseline approach only achieves a Precision of 49.37%. In contrast, DeepDir can reach 65.82%. In addition, the Recall and F-Measure values of DeepDir are higher than those of the baseline approach by 18.03% and 17.30%, respectively. From the perspective of F-Measure, the maximum improvement of DeepDir against the baseline approach is up to 22.83% (i.e., the JFace API specification).

Table III also presents the average comparison results between the two approaches. We can see that the baseline approach only achieves an average Precision of 31.10%. In contrast, DeepDir can achieve an average Precision of 51.62%. The average improvement of DeepDir against the baseline approach is 20.52%. In terms of the average Recall, the baseline approach achieves 73.21%. Meanwhile, DeepDir achieves 80.90%. From the perspective of the average F-Measure, the baseline approach only obtains 41.42%. In contrast, DeepDir can reach 62.20%. On average, DeepDir improves the baseline approach by 20.78% in terms of F-Measure. Hence, DeepDir significantly improves the performance of API directive detection compared against the baseline approach.

After demonstrating the effectiveness of DeepDir, we explain the underlying reasons why DeepDir performs better than the baseline approach. First, DeepDir is a deep learning based approach, and it can accurately learn the semantic differences of directives and non-directives based on the training set. In contrast, the baseline approach only relies on a set of syntactic patterns, and does not leverage the valuable information (class labels) of other sentences. Second, the nature of directives makes them extremely scarce in API specifications, and DeepDir employs an over sampling strategy to effectively deal with the imbalance between directives and non-directives. Third, DeepDir embeds the words in API specification into vectors to learn their distributional representation. In such a way, the semantic of each word can be learned.

Conclusion. Compared against the baseline approach, DeepDir is a more accurate approach to automatically detect API directives in API specifications.

C. Investigation to RQ3

RQ3: What is the performance of DeepDir when conducting the cross-project prediction?

Motivation. It is labor intensive to manually annotate API directives for new API specifications. To overcome this problem, the cross-project prediction may be employed to evaluate the performance of the proposed approaches when there is lack of annotated data. The cross-project prediction trains DeepDir using the annotated data from *source projects* and predicts API directives for *target projects*. Generally, the characteristics of the source projects and the target projects have different distributions, making the cross-project prediction more difficult than the within-project prediction.

TABLE IV
DETAILED RESULTS OF DEEPDIR IN CROSS-PROJECT PREDICTION

Source	Target	Precision	Recall	F-Measure
2+3	1	23.14%	50.21%	31.88%
1+3	2	52.97%	44.97%	48.07%
1+2	3	65.46%	53.70%	58.98%

1: Java, 2: JFace, 3: commons.collections

Approach. There are three API specifications in the publicly available annotated API directive corpus. To measure the performance of DeepDir in the cross-project prediction, we regard any two of the three API specifications as the source projects and the rest one as the target project. Specifically, we first trains DeepDir based on two API specifications. Then, we use the trained DeepDir to predict API directives in the rest API specification in the corpus. In such a way, we can evaluate the performance of DeepDir when conducting the cross-project prediction.

Results. Table IV shows the detailed results of DeepDir when conducting the cross-project prediction. The Java, JFace, and commons.collections API specifications are represented by *1*, *2*, and *3* in the table respectively. When training DeepDir on the Java and JFace API specifications and predicting on the commons.collections API specification, DeepDir achieves a F-Measure of 58.98%. Meanwhile, when treating the Java and commons.collections API specifications as source projects and regarding the JFace API specification as the target project, DeepDir achieves Precision, Recall, and F-Measure of 52.97%, 44.97%, and 48.07% respectively. Similar as the within-project prediction, DeepDir achieves the worst results in the Java API specification in the corpus when conducting the cross-project prediction. For example, when testing on the Java API specification and training on the other two API specification, DeepDir achieves a F-Measure of 31.88%.

The cross-project prediction is a more challenging task, since the source projects have different characteristics with the target projects. Hence, the performance of DeepDir in the cross-project prediction is not as good as in the within-project prediction. From the results of DeepDir in the cross-project prediction, we can see that the semantics differences between directives and non-directives learned from a project can be used to predict a different project. Thus, DeepDir is able to capture the common characteristics of API directives across different API specifications, and DeepDir is applicable in the cross-project prediction.

Conclusion. Even though the cross-project prediction is more difficult, DeepDir still performs well. DeepDir can be used in the real scenarios.

VI. THREATS TO VALIDITY

In this section, we discuss the threats to validity, including the threats to internal validity and the threats to external validity.

A. Threats to Internal Validity

The threats to internal validity include the potential errors or biases in the implementation and the experiments. We have employed the code review process to examine our code to implement our approach. Meanwhile, we are to open our source code when the paper is published. In the experiments, we have already validated the effectiveness of DeepDir with the over sample component. In addition, we have also verified DeepDir to conduct the cross-project prediction. The experimental results reveal that DeepDir is solid and can be used in different scenarios.

B. Threats to External Validity

The threats to external validity are related to the generalizability of DeepDir. We have verified DeepDir over the publicly available corpus. This corpus is constructed by following a strict and standard case-study protocol, so its quality can be guaranteed. It contains three API specifications with more than 85 thousand sentences in total, so it is large enough to verify DeepDir. Experimental results show that DeepDir significantly outperforms the state-of-the-art approach. It is unknown what is the performance of DeepDir over other API specifications. In the future, we plan to introduce more API specifications to validate DeepDir.

VII. Related Work

In this section, we introduce three related work, i.e., content analysis in API documentation, reference recommendation in API documentation, and deep learning in software artifacts.

A. Content Analysis in API documentation

Content analysis aims to explore what important information is contained API documentation. Maalej and Robillard study the nature and patterns of knowledge types in API documentation, and they manually develop a taxonomy of knowledge containing 12 knowledge types [1]. Monperrus *et al*. focus on one of the most important knowledge types, i.e., directive, and further empirically study and extensively discuss 23 kinds of directives [6]. Dekel *et al*. implement a tool *eMoose* to associate directives with methods [7], [29]. Zhou *et al*. try to detect directive related defects by analyzing API documentation and source code [30]. Zhong and Su detect API documentation errors by formulating a class of inconsistencies [31].

The above-mentioned studies have stressed the importance of directives and motivated us to propose an approach to automatically detect directives in API documentation.

B. Reference Recommendation in API documentation

Reference recommendation aims to actively and accurately discover desired information in API documentation for developers. Petrosyan *et al*. try to recommend tutorial fragments explaining APIs for developers [2]. Jiang *et al*. propose a more accurate approach to improve the performance of API fragment recommendation [24]. Furthermore, Jiang *et al*. also propose an unsupervised approach to tackle the shortcomings of supervised approaches [3]. Robillard *et al*. propose *Krec* to detect and recommend API documentation fragments for APIs [5]. Dagenais and Robillard recommend adaptive changes for API documentation evolution using traceability links [32]. Some other studies also try to recommend code samples to enrich API documentation [33], [34], [35].

Unlike the studies of recommending API documentation fragments or adaptive changes for APIs, we try to automatically detect directives in API specifications in this study.

C. Deep Learning in Software Artifacts

With the rapid growth of deep learning models, researchers have investigated their applications in software artifacts, especially source code. Gu *et al.* propose RNN based *DeepAPI* to help developers generate API usage sequences for queries in natural language [36]. Mou *et al.* conduct programming language representation by building a tree based Convolutional Neural Network (CNN) [37]. Hu *et al.* build a deep learning model to generate code comments [22]. Li *et al.* and Wang *et al.* employ CNN to predict software defects [38], [39]. In addition, Lam *et al.* combine deep learning models and information retrieval for bug localization [40].

In addition to source code, some studies explore the applications of deep learning models in software documents. Xu *et al.* employ CNN to predict linkable knowledge units in Stack Overflow [41]. Deshmukh *et al.* detect duplicate bug reports using neural networks [42].

Inspired by these studies with their promising results, we employ the Bi-LSTM network to automatically detect API directives in API specifications.

VIII. Conclusion and Future Work

API directive is one of the most important knowledge in API specifications. A survey conducted in this paper reveals that developers can really benefit from detecting and highlighting API directives. Existing approach only relies on syntactic patterns to detect API directives and lacks a deep semantic understanding. In this study, we propose a deep learning based approach, i.e., DeepDir, to automatically detect API directives. DeepDir first over samples directives in the training set. Then, it embeds the words of sentences into vectors and employs a Bi-LSTM network to learn the semantic differences between directives and non-directives. Finally, given a new sentence in API specification, the trained Bi-LSTM is used to predict whether it is a directive or not. Experiments over an annotated corpus show that, DeepDir significantly improves the state-of-the-art approach by up to 22.83% in terms of F-Measure. In addition, DeepDir can capture the common semantic characteristics of API directives across different API specification.

For the future work, we plan to improve DeepDir in the following directions. First, we try to consider the suggestions made by developers in the survey to further improve DeepDir. Second, we intend to evaluate DeepDir over more annotated API directive corpora.

REFERENCES

[1] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1264–1283, 2013.

[2] G. Petrosyan, M. P. Robillard, and R. de. Mori, "Discovering information explaining api types using text classification," *In Proceedings of the 37th International Conference on Software Engineering (ICSE 15)*, pp. 869–879, 2015.

[3] H. Jiang, J. X. Zhang, Z. L. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for apis," *In Proceedings of the 39th International Conference on Software Engineering (ICSE 17)*, pp. 38–48, 2017.

[4] L. Shi, H. Zhong, T. Xie, and et al., "An empirical study on evolution of api documentation," *In Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE 11)*, pp. 416–431, 2011.

[5] M. P. Robillard and Y. B. Chhetri, "Recommending reference api documentation," *Journal of Empirical Software Engineering*, vol. 20, pp. 1558–1586, 2015.

[6] M. Monperrus, M. Eichberg, E. Tekes, and et al., "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, pp. 703–737, 2012.

[7] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," *In Proceedings of the 31st International Conference on Software Engineering (ICSE 09)*, pp. 320–330, 2009.

[8] *http://commons.apache.org/proper/commons-collections/javadocs/api-2.1.1/org/apache/commons/collections/UnboundedFifoBuffer.html*.

[9] *http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html#removeElementAt(int)*.

[10] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," *In Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 15)*, pp. 33–42, 2015.

[11] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, pp. 618–643, 2010.

[12] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, pp. 68–75, 2015.

[13] *https://www.surveymonkey.com/r/23XP7X9*.

[14] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703–732, 2011.

[15] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *In Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS 13)*.

[16] S. Berlemont, F. Mamalet, and C. Garcia, "Blstm-rnn based 3d gesture classification," *In Proceedings of the International Conference on Artificial Neural Networks and Machine Learning*, pp. 381–388, 2013.

[17] S. Tang, Z. Y. Wu, and K. Chen, "Movie recommendation via blstm," *In Proceedings of the International Conference on Multimedia Modeling*, pp. 269–279, 2017.

[18] A. Ray, S. Rajeswar, and S. Chaudhury, "Text recognition using deep blstm networks," *In Proceedings of the Eighth International Conference on Advances in Pattern Recognition*, pp. 1–6, 2015.

[19] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," pp. 1532–1543, 2014.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[21] *https://www.tensorflow.org/*.

[22] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," *In Proceedings of the IEEE International Conference on Program Comprehension (ICPC 18), to appear*, 2018.

[23] S. Y. Jiang, A. Armaly, and C. Mcmillan, "Automatically generating commit messages from diffs using neural machine translation," *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 17)*, pp. 135–146, 2017.

[24] H. Jiang, J. X. Zhang, X. C. Li, and et al., "A more accurate model for finding tutorial segments explaining apis," *In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 16)*, pp. 157–167, 2016.

[25] X. Xia, D. Lo, E. Shihab, X. Y. Wang, and X. H. Yang, "Elblocker : Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[26] C. Zhang, J. Y. Yang, Y. Zhang, and et al., "Automatic parameter recommendation for practical api usage," *In Proceedings of the 34th International Conference on Software Engineering (ICSE 12)*, pp. 826–836, 2012.

[27] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The effects of over and under sampling on fault-prone module detection," *In Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pp. 196–204, 2007.

[28] H. Han, W. Y. Wang, and B. H. Mao, "Borderline-smote: a new over-sampling method in imbalanced data sets learning," *In Proceedings of the International Conference on Advances in Intelligent Computing*, pp. 878–887, 2005.

[29] U. Dekel, "Increasing awareness of delocalized information to facilitate api usage," *PhD Thesis Carnegie Mellon University*, pp. 1–284, 2009.

[30] Y. Zhou, R. H. Gu, T. L. Chen, and et al., "Analyzing apis documentation and code to detect directive defects," *In Proceedings of the 39th International Conference on Software Engineering (ICSE 17)*, pp. 27–37, 2017.

[31] H. Zhong and Z. D. Su, "Detecting api documentation errors," *In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA 13)*, pp. 803–816, 2013.

[32] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *IEEE Transactions on Software Engineering*, vol. 40, pp. 1126–1146, 2014.

[33] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," *In Proceedings of the 38th International Conference on Software Engineering (ICSE 16)*, pp. 392–403, 2016.

[34] J. Kim, S. Lee, S. Hwang, and S. Kim, "Enriching documents with examples: A corpus mining approach," *ACM Transactions on Information Systems (TOIS)*, vol. 1, p. 1, 2013.

[35] Y. C. Wu, L. W. Mar, and H. C. Jiau, "Codocent: Support api usage with code example and api documentation," *In Proceedings of 5th International Conference on Software Engineering Advances (ICSEA 10)*, pp. 135–140, 2012.

[36] X. D. Gu, H. Y. Zhang, D. M. Zhang, and S. Kim, "Deep api learning," *In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 16)*, pp. 631–642, 2016.

[37] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," *In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 16)*, pp. 1287–1293, 2016.

[38] J. Li, P. J. He, J. M. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," *In Proceedings of the International Conference on Software Quality, Reliability and Security*, p. 318328, 2017.

[39] S. Wang, T. Y. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," *In Proceedings of the 38th International Conference on Software Engineering (ICSE 16)*, pp. 297–308, 2016.

[40] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," *In Proceedings of the International Conference on Automated Software Engineering (ASE 16)*, pp. 476–481, 2016.

[41] B. W. Xu, D. H. Ye, Z. C. Xing, X. Xia, G. B. Chen, and S. P. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," *In Proceedings of the International Conference on Automated Software Engineering (ASE 16)*, pp. 51–62, 2016.

[42] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," *In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 17)*, pp. 115–124, 2017.