Automating Intention Mining

Qiao Huang, Xin Xia, David Lo, Gail C. Murphy

Abstract—Developers frequently discuss aspects of the systems they are developing online. The comments they post to discussions form a rich information source about the system. Intention mining, a process introduced by Di Sorbo et al., classifies sentences in developer discussions to enable further analysis. As one example of use, intention mining has been used to help build various recommenders for software developers. The technique introduced by Di Sorbo et al. to categorize sentences is based on linguistic patterns derived from two projects. The limited number of data sources used in this earlier work introduces questions about the comprehensiveness of intention categories and whether the linguistic patterns used to identify the categories are generalizable to developer discussion recorded in other kinds of software artifacts (e.g., issue reports).

To assess the comprehensiveness of the previously identified intention categories and the generalizability of the linguistic patterns for category identification, we manually created a new dataset, categorizing 5,408 sentences from issue reports of four projects in GitHub. Based on this manual effort, we refined the previous categories. We assess Di Sorbo et al.'s patterns on this dataset, finding that the accuracy rate achieved is low (0.31). To address the deficiencies of Di Sorbo et al.'s patterns, we propose and investigate a convolution neural network (CNN)-based approach to automatically classify sentences into different categories of intentions. Our approach optimizes CNN by integrating batch normalization to accelerate the training speed, and an automatic hyperparameter tuning approach to tune appropriate hyperparameters of CNN. Our approach achieves an accuracy of 0.84 on the new dataset, improving Di Sorbo et al.'s approach by 171%. We also apply our approach to improve an automated software engineering task, in which we use our proposed approach to rectify misclassified issue reports, thus reducing the bias introduced by such data to other studies. A case study on four open source projects with 2,076 issue reports shows that our approach achieves an average AUC score of 0.687, which improves other baselines by at least 16%.

1 Introduction

During the process of software development, developers frequently discuss how to resolve defects, what features to implement, the overall project plan, and many other points via various written communication channels, including mailing lists, issue repositories, and code review systems [1]–[5]. These channels keep track of developer discussions, easing the process of decision making and facilitating effective communication in a distributed project team [1], [5], [6]. The discussions are a rich source of information that can be used to build multiple kinds of support tools for developers [4], [7]–[9].

When developers participate in a communication channel, they may have different intentions. For example, the intention of developer A could be describing a bug, while the intention of developer B might be providing possible solutions. To help make developer discussions more accessible to analysis tools, Di Sorbo et al. proposed the concept of intention mining. Specifically, they proposed a taxonomy of intentions to classify sentences in developer discussions from mailing lists into six categories: *feature request*, *opinion asking*, *problem discovery*, *solution proposal*, *information seeking*, and *information giving* [10]. To build an automatic tool for classification, they manually extracted 231 heuristic linguistic patterns based on a collection of 1,077 manually labelled sentences from the Qt and Ubuntu mailing lists. They also showed how the determination of sentences in developer discussions related to four categories—*feature request*, *problem discovery*, *information seeking*, and *information giving*—can help generate method descriptions [10]. This taxonomy of intentions and the summarized linguistic patterns have also been applied to the analysis of user feedbacks collected from app reviews. For example, Panichella et al. [11] leveraged the linguistic patterns for the intentions of *information giving*, *information seeking*, *feature request*, and *problem discovery* to classify sentences in app reviews.

While Di Sorbo et al.'s taxonomy has been shown to be effective in analyzing developer discussion from mailing lists [10] and user feedbacks from app reviews [11], it is still unclear whether Di Sorbo et al.'s taxonomy of intentions can be generalized to discussions in issue tracking systems. In addition, it is unclear whether Di Sorbo et al.'s manually summarized heuristic linguistic patterns still work well for this new setting. In this paper, we first assess the generalizability of Di Sorbo et al.'s taxonomy on sentences appearing in comments recorded in issue tracking systems. We perform this assessment by randomly selecting 5,408 sentences from comments recorded in issue tracking systems of four large and popular projects hosted on GitHub (i.e., TensorFlow [12], Docker [13] (evolved to *Moby*), Bootstrap [14], and VS Code [15]). Then, we categorize these sentences manually, refining Di Sorbo et al.'s taxonomy during the process. Specifically, we merge *opinion asking* into *information seeking* since the ratio of sentences belonging to *opinion asking* is rather low (i.e., 1%) and they can always be classified as *information seeking*. We also add two additional intentions — *aspect evaluation* and *meaningless* – to provide

- *Qiao Huang is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. E-mail: tkdsheep@zju.edu.cn*
- *Xin Xia is with the Faculty of Information Technology, Monash University, Australia. E-mail: xin.xia@monash.edu*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Gail C. Murphy is with the Department of Computer Science, University of British Columbia, Canada. Email: murphy@cs.ubc.ca*
- *Xin Xia is the corresponding author.*

coverage of all sentences.

Next, to extend Di Sorbo et al.'s manual process, we propose a deep learning-based approach to categorize sentences automatically and more accurately into different categories of intentions compared to Di Sorbo's previous heuristic approach. Our approach first learns a matrix representation of a sentence using word embedding [16]. Next, our approach builds a convolutional neural network (CNN) to classify new sentences. CNN requires an initial setting of multiple hyperparameters; previous studies [17]–[19] have shown that tuning of hyperparameters is critical for a prediction model to achieve good performance. However, this tuning is tedious, time-consuming and requires expert knowledge. To overcome these limitations that can restrict the use of the approach, we propose an automatic hyperparameter tuning approach, along with an ensemble learning component to improve accuracy. Since CNN is computationally expensive and requires a long training time, we also integrate batch normalization [20] to substantially reduce the training time, making the approach more applicable in practice.

To evaluate the performance of our approach, we compare our approach with Di Sorbo et al.'s heuristic linguistic pattern approach [10]. We perform experiments on both our collected data (i.e., comments in issue reports) and Di Sorbo et al.'s data (i.e., emails in mailing lists). The experimental results show that our approach achieves an average accuracy of 0.8, which improves on Di Sorbo et al.'s approach by 91%. One might question whether our *enhanced* deep learning strategy is necessary to achieve this improvement. To investigate this question, we compare our approach against three other automated approaches that have been used to classify sentences for various purposes: Kim's CNN for sentence classification [21], Gu and Kim's natural language processing approach [22], and traditional text classification using bag-of-words features [23] and hyper-parameter tuning strategies [24], [25]. Our experiments demonstrate that our approach improves on these techniques by an average improvement of at least 13% in terms of accuracy. In terms of efficiency, in our experiments, our approach requires about half an hour to automatically tune the hyperparameters and train multiple CNNs for ensemble learning.

A substantial threat to the generalizability of our approach is bias in the labelling process used to train the neural network. To investigate this bias, we conducted a user study with 11 professional developers to check on the appropriateness of intention categories that are produced by our approach when it is applied to 140 sentences from issue tracking systems of four other projects (i.e., Three.js, Ruby on Rails, OpenCV and Scikit-Learn). We found that on average across all categories, 89% of the classification results are accepted by the majority of the developers, and 73% of the classification results are accepted by all developers.

Finally, we apply our approach in a specific automated software engineering task, namely rectifying misclassified issue reports. Herzig et al. [26] found that issue reports are often misclassified – many are marked as bug reports when they are actually feature requests and vice versa. This misclassification would introduce bias and threaten the external validity of any study that builds on such data – e.g., defect prediction studies [27], [28]. Given an issue report classified as bug report, our approach searches for sentences that are likely to belong to feature requests and reclassifies this issue report as a feature request if it contains at least one such sentence. A case study on four open source projects with 2,076 issue reports shows that our approach achieves an average AUC score of 0.687, which improves a number of baselines proposed by Antoniol et al. [29] by at least 16%. Different from the existing baselines, our approach not only rectifies a misclassified issue report, but also shows the reason why it should be categorized as such. This indicates that our approach can serve as a support tool to help in rectifying misclassified issue reports.

The main contributions of this paper are:

1) We assess the generalizability of Di Sorbo et al.'s taxonomy of intentions on discussions in issue tracking systems. We proceed to refine their taxonomy by merging two intentions into one category, and adding two new intentions.
2) We apply CNN and extend it to better classify sentences into intention categories. We integrate batch normalization with CNN to boost the training speed by at least 10 times. We improve the accuracy of CNN by at least 10% with automatic hyperparameter tuning and ensemble learning.
3) We show our approach achieves a good performance in terms of accuracy and F1-score, which outperforms both Di Sorbo et al.'s approach and other automated approaches for sentence classification by a substantial margin.
4) We conduct a user study with 11 professional developers and we find that 89% of the classification results are agreed by the majority of developers.
5) We conduct a case study to show how our approach can be applied to improve an automated software engineering task.

Paper Organization. The remainder of the paper is organized as follows. We present background and related work in Section 2. We introduce our taxonomy of intentions in Section 3. We elaborate the preliminaries of deep learning in Section 4. We describe the overall framework and technical details of our approach in Section 5. We present our experimental setup and results in Section 6. We present the user study of our classification results in Section 7. We apply our approach to improve an automated software engineering task in Section 8. We discuss how to identify patterns in a sentence the explain CNN's classification result and discuss threats to validity in Section 9. We conclude the paper and mention future work in Section 10.

2 Background and Related Work

2.1 Intention Mining

Although Di Sorbo et al. [10] proposed the concept of intention mining from developer discussions, they did not explicitly define what is *intention*. In Oxford English dictionary, the word *intention* is defined as "*A thing intended; an aim or plan*". In this paper, we define the concept of intention as the purpose underlying an expression by a developer in a discussion. For example, in an issue report, the intention of some comments is to describe a bug, while the intention of some other comments is to provide possible solutions.

There are various potential applications of the intention mining results. For example, we can leverage the identified *information seeking* sentences to summarize frequently asked questions in a project, or we can recommend the most wanted features to the project manager based on identified *feature request* sentences.

In practice, the comments posted by the same developer can contain a mix of intentions. For example, a developer posted the following comment for Issue #7654 on Tensorflow[1]:

> *Currently the initial load / full page refresh of tensorboard starts on the scalars tab with all runs toggled on for rendering. If there are a lot of runs in the logdir, the browser hangs for a noticeable period of time while all of the charts for the runs are rendered. I suggest making the initial load have runs off and letting the user opt in / turn on the desired set of runs.*

These three sentences have different intentions: the first sentence introduces how tensorboard is initialized (*information giving*); the second sentence reports a bug about the low performance in rendering (*problem discovery*); the final sentence requests a new feature to allow user customizing the set of runs for initial loading (*feature request*). Di Sorbo et al. first proposed the problem of intention mining from developer discussions [10]. This earlier work has the following limitations:

1) It is unclear whether the six intention categories are comprehensive. Di Sorbo et al.'s study focused on only one kind of developer discussions; its applicability to other discussions, such as ones that occur in issue tracking systems, is unknown.
2) The approach proposed by Di Sorbo et al. requires a large amount of manual effort; all of the 231 heuristic linguistic identification patterns are generated manually. Moreover, since these patterns are generated from only 1,077 sentences, it is unclear whether the patterns discovered on this data generalize beyond mailing lists. If they cannot be generalized, the approach is limited in its applicability as creating patterns manually requires substantial effort.

Our paper aims to reduce the limitations of this earlier work by: (1) assessing the generalizability, and potentially refining, Di Sorbo et al.'s taxonomy on another kind of developer discussion, and (2) proposing a more accurate and automated intention mining approach.

While Di sorbo et al. focused on intention mining from developer discussions (i.e., discussions in mailing lists), there are also a number of studies on intention mining from user feedback, especially from app reviews [11], [22], [30]–[32]. These studies leveraged intention mining as a fundamental component to support more complicated tasks. Panichella et al. refined Di Sorbo et al.'s taxonomy of intentions to classify sentences of app reviews into five intention categories (i.e., *feature request*, *problem discovery*, *information seeking*, *information giving*, and *others*). They manually extracted 246 linguistic patterns from 500 app reviews [11] and most of these patterns are overlapping with those proposed by Di Sorbo et al., which demonstrates that the taxonomy of intentions and the linguistic patterns proposed by Di Sorbo et al. are also effective for app reviews. In a later work, Di Sorbo et al. proposed SURF to summarize app reviews for software change recommendation [30]. Palomba et al. proposed ChangeAdvisor, an approach that analyzes the intentions of sentences contained in user reviews to extract potential app changes, and recommend code components impacted by the suggested changes [31]. Both SURF and ChangeAdvisor leveraged the intention classifier proposed by Panichella et al. [11].

Gu and Kim proposed SUR-miner, which first classified sentences of app reviews into five intention categories—*aspect evaluation*, *praises*, *function requests*, *bug report*, and *others*—and then extracted aspects (e.g., specific set of options/suggestions) in sentences to understand what parts of the app are loved by users [22]. In the intention mining phase, given a sentence, their approach extracted and considered five types of features: character n-gram, trunk word, POS tag, parsing tree and semantic dependence graph. Their approach built a max entropy classifier based on these features. In this paper, we also compare our approach with their approach.

The approach we describe in this paper differs from these studies in focusing on mining intentions from developer discussions rather than from app reviews, which are more representative of an input from users to the developers and are typically not in discussion form. In this paper, we focus on intention mining in developer discussions, taking inspiration from similar work in app reviews, and we leave potential applications of the results to future work.

2.2 Classification of Software Artifacts

In general, intention mining is a classification of software artifacts into categories that could be relevant for software evolution. Aside from mining developer discussions, there are also many other studies which apply classification techniques to analyze other types of software artifacts. For example, a large number of studies applied different classification techniques to classify different types of bug reports [33]–[37]. Antoniol et al. [29] used text classification techniques to predict whether a change requests is a bug report or a feature request. Chaparro et al. [38] proposed an automated approach to detect the absence (or presence) of expected behavior and steps to reproduce in descriptions of bug reports. Petrosyan et al. [39] proposed an approach to discover tutorial sections that explain a given API type. Their approach classifies fragmented tutorial sections using supervised text classification based on linguistic and structural features. Hou and Mo [40] applied Naive Bayes to categorize the content of API discussions in online forums. Prasetyo et al. [41] applied texting mining techniques to classify microblogs into two categories: relevant and irrelevant to engineering software systems. Thung et al. [42] combined both design and network metrics to condense a reverse-engineered class diagram by predicting if a class is important or not. Our work is different from these studies, since we focus on mining intentions from developer discussions; and thus it complements the existing body of work on classifying software artifacts.

1. https://github.com/tensorflow/tensorflow/issues/7654

2.3 Deep Learning in SE

Recently, a number of studies have explored the possibility of applying deep learning techniques to software engineering, e.g., bug localization [43], [44], defect prediction [45], code completion [46], software community question retrieval [47], [48], and API search [49].

Deep learning aims to model high-level abstractions in data by building neural networks with multiple layers [50], and it has following advantages: (1) it largely reduces the cost on feature engineering by automatically learning advanced features from raw data; and (2) it can achieve very good performance and significantly outperform other solutions in multiple domains by a substantial margin [16], [51]–[53].

On the other hand, deep learning approaches also have the following disadvantages: (1) it is computationally expensive, and the training process can cost several weeks [54] for a complicated network learned from massive training data even with multi-GPU accelerating; and (2) the setting of hyperparameters has a large effect on the performance of the model, and suitable setting of these hyperparameters requires much experience or expert knowledge.

Considering the promising results of deep learning, in this paper, we aim to leverage deep learning to mine intentions from developer discussions, and we also aim to overcome the weaknesses of deep learning models by reducing model building time and automatically tuning hyperparameters.

3 Taxonomy of Intentions

Di Sorbo et al. proposed a taxonomy of intentions, which contains six categories: feature request, opinion asking, problem discovery, solution proposal, information seeking, and information giving [10]. In this section, we investigate the comprehensiveness of Di Sorbo et al.'s intention categories, considering whether the categories cover different kinds of sentences posted in a different kind of developer discussion in different projects. We choose Di Sorbo et al.'s taxonomy as a starting point because they are the first who proposed the problem of intention mining. Besides, they have shown that their approach based on the taxonomy can be successfully applied for source code re-documentation by extracting method descriptions from developer discussions. If their taxonomy is still effective for developer discussions in issue tracking systems, then we can directly apply their taxonomy and focus on improving the efficiency and accuracy of sentence classification.

We collected developer discussions from issue tracking systems of four large-scale projects on GitHub, namely TensorFlow, Docker, Bootstrap, and VS Code. Although there are other issue tracking systems like Bugzilla and JIRA, we chose to focus on projects hosted on GitHub. The four projects belong to different application domains and have a large number of issue reports. Specifically, TensorFlow is a software library for numerical computation using data flow graphs; Docker is a tool providing container technology; Bootstrap is a front-end framework for web development; and VS Code is a source code editor for building and debugging modern web and cloud applications. Besides, since GitHub is based on social coding, it also provides several metrics to show the popularity of a project. For example, the two metrics *stars* and *watchers* represent how many people are interested in this project and are continuously following it. The metric *forks* represents how many projects are developed based on this original project. Based on the recorded large number of stars/watchers/forks of the four selected project, the projects that we selected are popular in the open source community; this helps us collect sentences posted by different developers with a diversity of writing styles. Finally, Table 1 presents statistics of these four projects. The columns are project name, number of stars/watchers/forks, programming language used in the project, total number of issues, number of issues we analyzed, number of sentences we manually labelled, and time period of the issues we analyzed.

The first author first conducted a *preliminary exploration* with 200 issue reports randomly selected from TensorFlow. The purpose of this exploration is to investigate whether Di Sorbo et al.'s taxonomy of intentions and their linguistic patterns show promise to be effective for classifying sentences from developer discussions in issue tracking systems. We chose TensorFlow since: 1) TensorFlow is one of the most popular deep learning frameworks, and our proposed approach is based on deep learning; 2) the first author was learning how to use TensorFlow when the study began, thus making it easier to understand the intentions of developer discussions in TensorFlow. Since there are a large number of comments in these 200 issue reports, we extracted the paragraphs of comments, and performed a stratified sampling of the paragraphs (i.e., more paragraphs would be randomly selected from issue reports with more comments). Then, we split the sampled paragraphs into sentences with the Stanford CoreNLP toolkit [55]. Next, we used regular expression to remove source code and stack traces – since they do not contain user intentions. In this way, we got 2,256 sentences in total. Note that our sentences were sampled from the randomly selected 200 issue reports, instead of directly sampling from all issue reports. We chose the sampling strategy based on a limited number of issue reports because the *context* of a sentence is often needed to understand the likely intention behind the sentence. For example, the sentence "*if you download tensorflow from github instead of 'git clone', you will not meet this*" in TensorFlow Issue #4312 seems to belong to *information giving*. However, since it was extracted from an issue report about a configuration error, it would be easy to understand that the word 'this' in the sentence refers to the error and this error could be avoided if we download tensorflow using 'git clone'. Thus, this sentence should belong to *solution proposal*. From this example, we can see that it can require substantial effort if sentences are sampled from all issue reports to achieve reliability in intentions determination, as many issue reports may need to be consulted to determine the content of sentences.

We first applied Di Sorbo et al.'s linguistic patterns to classify these sentences extracted from the 200 issue reports of TensorFlow, and we found that about 55% of all sentences could not be classified (i.e., they could not be matched with any patterns). Although some sentences might be meaningless (e.g., "*Sorry for my late reply*"), it is not likely that more than half of the comments in an

TABLE 1
Statistics of the analyzed projects (recorded on April 13rd, 2017).

Project	Stars	Watchers	Forks	Language	# Total Issues	# Analyzed Issues	# Labeled Sentences	Time Period
TensorFlow	55,396	4,993	26,380	C++/Python	6,075	200	2256	Nov 2015 - Jan 2017
Docker	43,101	3,213	12,797	Go	15,652	100	1216	Jun 2013 - Dec 2016
Bootstrap	109,884	6,839	50,557	JavaScript	14,885	100	1123	Sep 2011 - Dec 2016
VS Code	26,771	1,495	3,664	TypeScript	23,933	100	813	Nov 2015 - Mar 2017

TABLE 2
The refined taxonomy of intentions based on Di Sorbo et al.'s work. The two new categories are highlighted in bold.

Category	**Decription**	**TensorFlow**		**Docker**		**Bootstrap**		**VS Code**	
		Num	Ratio	Num	Ratio	Num	Ratio	Num	Ratio
Information Giving (IG)	Share knowledge and experience with other people, or inform other people about new plans/updates (e.g. "The typeahead from Bootstrap v2 was removed.").	543	24%	287	24%	305	27%	194	24%
Information Seeking (IS)	Attempt to obtain information or help from other people (e.g. "Are there any developers working on it?").	359	16%	263	22%	192	17%	148	18%
Feature Request (FR)	Require to improve existing features or implement new features (e.g. "Please add a titled panel component to Twitter Bootstrap.").	213	9%	128	11%	107	10%)	88	11%
Solution Proposal (SP)	Share possible solutions for discovered problems (e.g. "I fixed this for UI Kit using the following CSS.").	274	12%	129	11%	109	10%	59	7%
Problem Discovery (PD)	Report bugs, or describe unexpected behaviors (e.g. "the firstletter issue was causing a crash.").	400	18%	127	10%	132	12%	103	13%
Aspect Evaluation (AE)	Express opinions or evaluations on a specific aspect (e.g. "I think BS3's new theme looks good, it's a little flat style.").	349	16%	184	15%	159	14%	159	20%
Meaningless (ML)	Sentences with little meaning or importance (e.g. "Thanks for the feedback! ").	118	5%	98	8%	119	11%	62	8%

issue report are meaningless. Thus, the result show that a lot of information would be missed when using Di Sorbo et al.'s linguistic patterns. Besides, we found that while many sentences were classified as *solution proposal* or *feature request* by linguistic patterns, they did not seem to belong to these categories. For example, the sentence *"Currently there is no way to do this with 2d convolutions as mentioned by @alphaf52."* in TensorFlow Issue #1136 was classified as *solution proposal* while it should belong to *information giving*. Thus, the result show that while the majority of Di Sorbo et al.'s taxonomy of intentions and their linguistic patterns are still effective for developer discussions from issue tracking systems, refinement is needed to cover more sentences in this kind of developer discussions.

Next, to increase the generalizability of our study, we randomly selected 100 issue reports from the other three projects, using the same selection approach as for TensorFlow. The high cost of manually labelling sentences, and the context required for manually inspection, limits the overall number of issue reports, paragraphs and sentences considered. The approach taken also enables the investigation of the impact of different amount of training data on the effectiveness of our approach when cross-project prediction is employed. In total, we collected 5,408 sentences from 500 issue reports in these four projects.

Then, we created one card for each sentence, where the card contained the natural language description of the sentence, and the context of the sentence, specifically the full paragraph in which the sentence appeared and the corresponding issue number. We used two iterations of a card sorting approach [56] to label these sentences.

Iteration 1. We first randomly picked 1,000 sentences from the 5,408 sentences in the dataset. The first two authors manually and independently categorized these sentences based on the six intention categories proposed by Di Sorbo et al. [10]. If an author considered the sentence could not be categorized into any of the six categories, it was set aside for further discussion. Next, the first two authors and an invited post-doc (who is not a co-author of this paper) worked together to discuss the disagreements in the labeling process and cases that could not be categorized into the six categories. We found:

1) Only 11 (1.1%) sentences were manually classified as *opinion asking* by any one of the two labelers. We also applied Di Sorbo et al.'s linguistic patterns to classify the 1,000 sentences in Iteration 1 and only 7 (0.7%) sentences were classified as *opinion asking*. This finding is consistent with Di Sorbo et al.'s work, where they found that among all the 1,077 sentences extracted from mailing lists, only 17 sentences (1.6%) were classified as *opinion asking*. On the other hand, sometimes it is difficult to clearly distinguish these two categories. For example, the sentence *"What do you think about the Zeros solution?"* in TensorFlow Issue #783 can be categorized into either category. Such ambiguity also exists in Di Sorbo et al.'s manually summarized linguistic patterns for these two categories. For example, they identified the pattern *"Is [something] you prefer?"* for the *opinion asking* category, while they also identified the pattern *"Which [something] do you prefer/favor/like?"* for the *information seeking* category. Considering the low ratio of sentences belonging to opinion asking and the fact that *opinion asking* is, in essence, a sub-category of *information seeking*, we merge *opinion asking* into the *information seeking* category in this study to avoid the ambiguity.
2) Some sentences express opinions or evaluations on a specific aspect, and cannot to be categorized into the six categories. The evaluated aspects can vary; for instance, sometimes the evaluated aspect is a bug, a feature, a project, a developer or something else. For example, in the sentence *"But I think it's cleaner than my old test, and I prefer a non-JS solution personally."* in Bootstrap Issue #1935 expresses the user's preference for a certain type of solution. Thus, we create a new category *aspect evaluation* to complement the intention categories. Sentences belonging to *aspect evaluation* can help us better

understand user preferences and viewpoints. Moreover, the category of *aspect evaluation* is also inspired by Gu and Kim's study [22]. For example, we found that some feature requests have been discussed for years, but developers are still not willing to implement them. We can extract relevant *aspect evaluation* sentences to summarize the positive and negative views of different developers, which may be used to help maintainers better understand what users want and what developers are concerned about.

3) Some sentences in developer discussions are meaningless for bug resolution or feature implementation, e.g., "*Thanks for your reply*". In this study, we create a new category *meaningless* to include these meaningless sentences. Although Di Sorbo et al.'s linguistic patterns do not cover such sentences, this *meaningless* category is necessary for our approach, since any input sentence would finally be assigned to one intention category in our approach.

We use Fleiss Kappa [57] to measure the agreement between the two labelers. There are 295 sentences (29.5% of all sentences) with disagreement between the two labelers, and the Kappa value is 0.64, which indicates substantial agreement. In total, there are 172 (17.2%) sentences that are not able to be categorized by at least one labeler into the six categories of Di Sorbo et al.'s taxonomy. About half of these sentences are meaningless and most of the other sentences belong to the case we described for the *aspect evaluation* category.

Among the sentences that are disagreed with the two labelers, most of them belong to the case where one labeler chose *information giving* category while the other labeler chose *solution proposal* or *feature request* category. Note that Di Sorbo et al. defined the description of *information giving* as "*linguistic patterns exploited to inform/update other users about something*". This definition is a bit vague since the sentences of some other categories (e.g., *solution proposal* or *aspect evaluation*) are also to inform other users about something. To improve the consistency during the manual labeling process, if a sentence seems to belong to multiple categories, we always prefer the other categories instead of *information giving*. Thus, our definition of *information giving* emphasizes more about the information with less subjectivity (e.g., knowledge, project plan), and the sentence should state the explicit kind of information.

Besides, a feature is more related to a new proposed functionality, while an aspect is more related to an existing characteristic of a project. When developers propose a solution or a feature request, the style they write their sentences may be similar to *aspect evaluation* sentences (e.g., "*I think configure need an option to decide target environment*" in TensorFlow Issue #4312 (*feature request*) and "*I think the best solution is to explicitly add a second derivative function in Python*" in TensorFlow Issue #4174 (*solution proposal*)). In such case, we prefer *solution proposal* and *feature request* instead of *aspect evaluation*.

In summary, at the end of this iteration, we find that the majority of categories in Di Sorbo et al.'s taxonomy of intentions are still applicable for developer discussions in issue tracking systems. We have only made a few adjustments. The final seven intention categories identified at the end of this iteration are shown in Table 2.

Iteration 2. The first two authors then independently labelled the remaining 4,408 sentences of the four projects into the seven categories as shown in Table 2. There are 621 sentences (14.1% of all sentences) with disagreement between the two labelers, and the overall Kappa value in this iteration is 0.78, which is higher than that of the first iteration, since the two labelers already had some experience to understand different intentions. Also, in this iteration, all sentences are successfully categorized with the new taxonomy of intentions. After completing the manual labeling process, the two labelers and another post-doc worked together to discuss their disagreements to reach a common decision.

During the labeling process, we faced the following challenges:

1) While the intentions of most sentences can be easily decided by reading the sentence itself or the context sentences in the paragraph, 0.8% of the sentences require labelers to carefully consider more sentences in the context paragraphs or comments posted by other developers. These sentences are the major cause of the disagreements between the two labelers. To address this challenge, the two labelers and another post-doc read the context paragraphs, and sometimes even the whole issue report, to make the final decision.
2) 1.3% sentences can be categorized into more than one intention categories simultaneously. To address this challenge, the two labelers and another post-doc discussed these sentences and decided the most suitable intention category based on agreement.

Table 2 shows the categories of the labelled sentences and the distribution of sentences for the four projects considered in our study. Note that the distribution of sentences varies in different projects. For example, TensorFlow has the highest proportion of sentences belonging to *problem discovery*. One possible reason is that Tensorflow is still at an early stage of development with rapid release cycles.

4 Background on Deep Learning

Our approach uses deep learning. In this section, we introduce background material on representing sentences using word embedding and the CNN architecture for sentence classification needed to understand the approach we introduce.

4.1 Sentence Representation

Traditional text classification techniques are usually based on a Vector Space Model (VSM) using bag-of-words features. In VSM, each word is treated as a discrete atomic symbol, and a sentence is represented as a vector of these symbols. Such an encoding cannot provide useful information regarding relationships that may exist between individual symbols. For example, while both *Maven* and *Ant* are Java build tools, we cannot calculate their "distance" using the encoding of discrete atomic symbols to evaluate whether they are semantically close to each other. Another downside of representing words as unique, discrete symbols is that the data becomes sparse. For example, if a vocabulary contains millions of unique words, then a sentence with a few words has to be represented by a vector with a very large dimension size while most values in the vector are just zero.

To address the issue of identifying semantically similar words, there are other techniques like latent semantic indexing (LSI) [58], probabilistic latent semantic analysis (pLSA) [59], and latent Dirichlet allocation (LDA) [60] that are able to map a document to a probabilistic distribution of different topics, and each topic is also represented by a probabilistic distribution of different words. In general, these techniques are called *topic modelling*, and the "topics" produced by these techniques are clusters of words that usually appear in similar context. However, topic modelling cannot be directly used for classification, since its original purpose is to discover the hidden semantic structures (i.e., different topics) from a large collection of unlabeled documents. Topic modelling is more widely used for information retrieval. However, some previous studies [61], [62] have shown that older and simpler IR technique (e.g., VSM), can perform better than more complicated IR techniques (e.g., LDA) for certain applications like bug localization.

Recently, a number of neural-network-based approaches [16], [63], [64] have been proposed to represent each word by a low dimensional vector called "word embedding". Word embedding depends on the distributional hypothesis [65], which states that words in the same context within a sentence tend to share semantic meaning. Thus, semantically similar words (e.g., *Maven* and *Ant*) would be close in the embedding space and we can easily evaluate the distance between them by calculating the cosine similarity of their corresponding embedding vectors. Word embedding has been successfully applied in software engineering tasks [44], [47]. In this paper, to learn word embedding, we follow these previous studies to leverage Mikolov et al.'s Skip-gram model [16], which is popular for its simplicity and efficiency during training.

Using word embedding, we represent a sentence by a $L \times D$ matrix, where L represents the maximum length of all sentences in the training dataset, and D represents the dimension size of embedding space. Note that D is a hyperparameter and all words would have the same dimension size when learning word embedding. In the matrix, the i^{th} row represents the embedding vector of the i^{th} word in the sentence. If a sentence's length l is less than L, we pad the last $L - l$ rows by zeros. We must pad since our CNN requires a fixed size of input matrix.

During the training phase, we use all sentences in the training dataset to learn word embedding, and we also store a dictionary which records the embedding vector of each unique word. In the prediction phase, given a new sentence, we first lookup the dictionary to retrieve the embedding vector for each word in the sentence. If a word cannot be found in the dictionary, it would be mapped to a special token in the dictionary called *UNK* (i.e., unknown word) and its embedding vector would be initialized as a vector of zeros. This implementation is based on the API provided by TensorFlow and it is also used by previous studies of deep learning in software engineering [49]. Finally, If the length of a new sentence is larger than L, we only keep the first L words so that the sentence is still represented by a $L \times D$ matrix.

4.2 CNN for Sentence Classification

In this section, we focus on introducing the basic idea and important technical details of Kim's Convolutional Neural Network (CNN) architecture for sentence classification [21]. The CNN consists of multiple layers, including an input layer, convolutional layer, pooling layer and output layer. The training of CNN follows an iterative process. In each iteration, data is passed through the layers. Each layer contains multiple neurons, which receive the output values of the previous layer and apply specific transformation to these values.

CNN has *parameters* and *hyperparameters*. The earlier are randomly initialized and will be learned during training, while the later are set by humans before training and their values will remain the same during the training process. For example, the *dimension size of word embedding* is one of the hyperparameters.

CNN starts with the input layer, which takes as input the matrix representation of a sentence. Here we denote this matrix as M. Then the convolutional layer receives the input matrix and performs convolution operation on it using different *filters*. Each filter is also a matrix, denoted as F, having the same width as the input matrix M, but varying in height. The purpose of each filter with height n is to capture the semantic information of each *n-gram* sequence (i.e., series of n consecutive words) in the sentence through convolution operation. Let $M_{i:i+n-1}$ represents the sub-matrix of M from the i_{th} row to the $(i+n-1)_{th}$ row, which contains the embedding vectors of n continuous words (i.e., n-grams). A convolution operation is the dot product between the filter matrix F and the n-grams sub-matrix $M_{i:i+n-1}$, adding by a bias value b_i, which is computed as follows:

$$o_i = F \cdot M_{i:i+n-1} + b_i \tag{1}$$

The filter is applied repeatedly to each possible n-grams in the input sentence with convolution operation and produces an output vector O of length $L - n + 1$, i.e., $O = [o_1, o_2, \ldots, o_{L-n+1}]$. The neural network will automatically learn appropriate parameters (i.e., values) in each filter matrix F and the bias value b_i through training, so that the output vector O can carry semantic information of different n-gram sequences in the input sentence. In practice, to enable the network to learn enough semantic information in different granularities (i.e., n-grams with different length), multiple filters with various heights are used. Note that the *combination of filter heights* and the *number of filters for each height* are two hyperparameters.

Figure 1 shows an example of the CNN architecture. The input sentence "*We need Java API support*" is represented as a matrix of size 5×4 (i.e., five words, each represented as an embedding vector of size 4). Then in the convolutional layer we perform convolution operations on the input matrix via different filters. Suppose we have a filter of size 2×4, then it would perform convolution operation with the sub-matrix of each 2-gram sequences in the input sentence, including "*We need*", "*need Java*", "*Java API*" and "*API support*", respectively. Thus, given an input matrix of size 5×4 and a

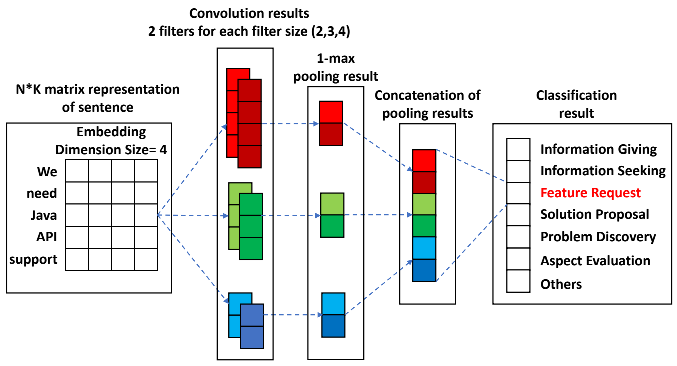

Fig. 1. An example of the CNN architecture.

filter of size 2×4 , we would get a vector of 4 output values after the convolution operations (i.e., marked by red color in Figure 1).

The convolutional layer is followed by a pooling layer which serves to progressively reduce the number of parameters that needs to be passed to the output layer and reduce the computation cost. Specifically, the pooling layer receives the output vector of each filter in convolutional layer and applies a *1-max pooling function* (i.e., extracts the maximum value) to the vector. Suppose we have m filters in total, then the pooling layer would receive m vectors and output m individual values. The pooling outputs of all filters are concatenated as a high-level feature vector and passed to the output layer.

The output layer contains n neurons and each neuron corresponds to one specific category for classification. Note that in Kim's CNN, the output layer only contains two neurons since it focuses on binary classification. When receiving the input vector (denoted as V, with size $m \times 1$) with m values, the output layer first perform linear transformation on it, computed as follows:

$$Y = W * V + B \tag{2}$$

Here W is a matrix of size $n \times m$ and B is vector with n values. Thus, the output Y is a vector with n values. The values in W and B are also automatically learned by neural network through training. Then, a softmax function is applied to normalize the values in Y so that each neuron's output can represent the probability of the input sentence belonging to one specific category. The softmax function is computed as follows:

$$Softmax\,(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \tag{3}$$

Suppose the input sentence belongs to the i_{th} category, then we denote G as a ground truth vector in which the i_{th} element value is 1 and all the other element values are 0. Then we use the cross-entropy function to measure the loss between the prediction result (i.e., the normalized vector Y produced by the output layer) and the ground truth (i.e., vector G). The cross-entropy function is computed as follows:

$$Loss(Y, G) = -\sum_{i}^{K} G_i log(Y_i) \tag{4}$$

The training process of CNN is an optimization task that proceeds in multiple iterations. In each iteration, the network predicts the labels of sentences in training data, and measures the *loss* between the prediction result and the ground truth label of each sentence. Then, the network will use backpropagation [66] to adjust the network parameters (e.g., values in the matrix of each filter). In the subsequent iterations, the training process of CNN tries to lower the average loss on the training data. Such process repeats many rounds until the average loss reaches convergence (i.e., its value has stabilized across iterations). In practice, we need to set a *learning rate* to decide how fast we would like the neural network to adjust the parameters. A higher learning rate can speed up the training process, but it may negatively impact the performance on classification. By default, Kim's CNN set learning rate as 10^{-4}.

5 Approach

In this section, we first present the overall framework of our approach. Then we introduce how we integrate batch normalization into CNN to improve its time efficiency, and how we improve the accuracy of CNN by automatic hyperparameter tuning and ensemble learning.

5.1 Overall Framework

Figure 2 presents the overall framework of our approach. It contains two phases: a training phase and a prediction phase. In the training phase, we first learn word embedding of all unique words in training data and represent each sentence by a matrix. Then we take as input the matrix representation of each sentence to train a CNN, which predicts the probability of an input sentence to belong to each of the intention categories. Since the CNN requires an initial setting of multiple hyperparameters, we propose an automatic hyperparameter tuning approach to search and select the appropriate values of two most important hyperparameters (i.e., dimension size of word embedding and number of filters for convolution operation) in a greedy way. The selected hyperparameters are then passed to the ensemble learning component, where we train multiple CNNs with different combination of filter heights, to achieve a more stable performance.

In the testing phase, for each new sentence, we first lookup the dictionary of word embedding learned in the training phase, and retrieve the embedding vector of each word in the sentence to represent the sentence by a matrix. Then we feed the sentence matrix into the ensemble learning component, and each CNN will predict the label of the sentence. Finally, the label predicted by the most CNNs will be chosen as the final prediction result.

5.2 Adapted CNN with Batch Normalization

To adapt Kim's CNN architecture to solve our problem (i.e., intention mining), we modify the output layer of their CNN from 2 neurons (i.e., binary classification) to 7 neurons (i.e.,

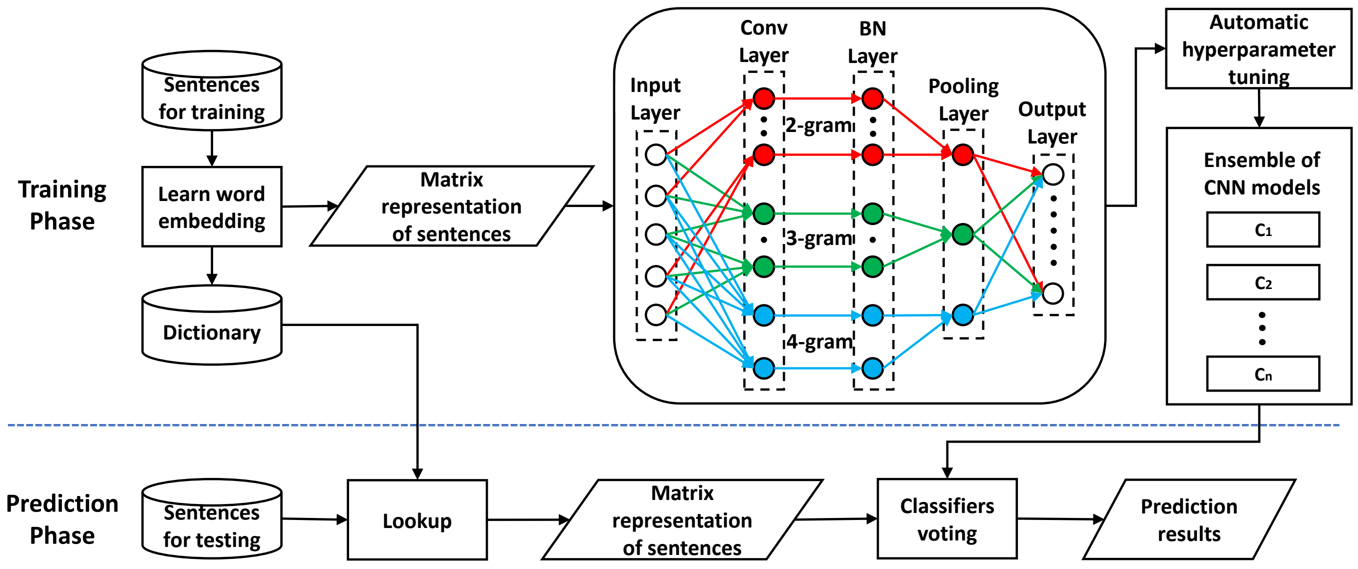

Fig. 2. Overall framework of our approach.

7-class classification, one for each intention category). By training appropriate parameters for each layer, the network would learn how to represent high-level semantic meaning of the input sentence using these parameters. And it also learns the relationship between this high-level representation and the corresponding intention. Given a new sentence, the trained network would also extract its high-level representation and identify its intention category even if there is no keywords matching between the new sentence and the sentences in training data.

One problem with deep learning is its high computation cost. Since a deep neural network has multiple layers with millions or even billions of parameters to learn during training [50], it is very time consuming if a practitioner wants to re-run the training process for hyperparameter tuning or debugging. Ioffe and Szegedy [20] pointed out that training deep neural network is complicated because the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rate and careful parameter initialization. They refer to this phenomenon as *internal covariate shift* and they proposed an algorithm called *batch normalization* to accelerate the training process by reducing internal covariate shift.

In our approach, we add a batch normalization (BN) layer between the convolutional layer and the pooling layer. In practice, the BN layer is usually added after the convolutional layer or the fully-connected layer (i.e. the layer that concatenates all pooling results in our architecture) [20]. We choose to add BN layer after the convolutional layer following Ioffe and Szegedy [20]. In general, batch normalization can reduce the number of iterations required to minimize the loss function. However, the more BN layers added, the more time is needed for a single iteration to run during the training process. Since our preliminary experiment has shown that adding one BN layer is already enough to drastically reduce the training time, we do not add another BN layer in our architecture to avoid potential negative impact on efficiency introduced by multiple BN layers. In future work, we plan to further investigate the impact of adding BN layers in different places.

By adding the BN layer, the output of the convolutional layer would be normalized before feeding to the pooling layer. For each filter of the convolution layer, suppose there are m output values, we refer to these m values as a *mini-batch*, represented by $X = (x_1, x_2, ..., x_m)$. We first calculate the mean value μ_X and variance σ_X^2 of the mini-batch X. Next, we normalize each output value x_i so that the distributions of the normalized values of $\hat{X}$ have the expected mean value of 0 and the variance of 1, i.e.,

$$\hat{x}_i \leftarrow \frac{x_i - \mu_X}{\sqrt{\sigma_X^2 + \epsilon}} \tag{5}$$

In the above equation, a small constant ϵ (10^{-3} by default) is added to avoid dividing by zero during normalization. Finally, we scale and shift the normalized values with a pair of parameters γ and β, and we feed each transformation result y_i to the corresponding neuron in the pooling layer, i.e.,

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{6}$$

In the above equation, the values of γ and β are automatically learned by the neural network during the training process. Note that γ and β are independent for each individual filter. If we have N filters, then the network will learn N pairs of γ and β. Using batch normalization, we are able to set a higher learning rate while the trained CNN model still performs good. In practice, we set learning rate to 10^{-3}, and it will automatically decay to 10^{-4} as the training progresses.

5.3 Automatic Hyperparameters Tuning

Training CNN requires initial setting of multiple hyperparameters (e.g., the number of filters). Previous studies [17]–[19] have shown that hyperparameter tuning is important for a prediction model to achieve better performance. However, manual hyperparameter tuning is tedious and requires much experience or expert knowledge. To address the above issue, we propose an automatic hyperparameter tuning approach.

Zhang and Wallace conducted a sensitivity analysis of (and wrote a practitioners' guide to) CNN for sentence classification [67]. They found that the dimension size of word embedding, the number of filters, and the combination of filter heights are the most important hyperparameters that have a large effect on performance, and should be tuned. Based on their findings, we focus on tuning of the first two hyperparameters. The third hyperparameter will be considered in the ensemble learning step (see Section 5.4).

In practice, it is impossible to enumerate the full search space of hyperparameters. In this paper, we first search and select the values of the first two hyperparameters in a greedy way. Specifically, we first enumerate the dimension size of word embedding from 64 to 320 with a step of 64, and train a CNN for each candidate dimension size. Note that during the enumeration process, all the other hyperparameters are set to their default values (i.e., the same values as those appearing in the published source code of Kim's CNN implementation based on TensorFlow [68]). Among all the candidate dimension sizes, we choose the one whose corresponding CNN achieves the minimum average loss on the training dataset. Similarly, we then enumerate the number of filters from 64 to 320 with a step of 64, and also choose the one whose corresponding CNN achieves the minimum average loss. Note that we do not increase these two hyperparameters further beyond 320 since most computation of our CNN is completed by GPU and the memory resource of our GPU is limited. We choose 64 as the step size since many previous studies (e.g., [52], [69]) in deep learning used multiples of 64 as values of these two hyperparameters.

5.4 Ensemble Learning

As for the hyperparameter of combination of filter heights, previous studies [20], [70]–[72] on CNN for image classification have shown that the ensemble of multiple CNNs with different combination of filter heights can achieve better and more stable performance. Thus, instead of choosing the "best" combination of filter heights, we simply combine multiple CNNs trained with different combinations of filter heights to predict new sentences together. Note that all these CNNs trained for ensemble learning use the automatically tuned dimension size of word embedding and number of filters by the approach described in Section 5.3.

In this paper, we consider ten combinations of filter heights, including (1,2,3), (2,3,4), (3,4,5), (4,5,6), (1,2,3,4), (2,3,4,5), (3,4,5,6), (1,2,3,4,5), (2,3,4,5,6) and (1,2,3,4,5,6). These combinations are produced by an enumeration from 1-gram to 6-gram with at least three different heights for one combination. Although there are still other combinations, we omit them since we need to control both the memory and time cost.

Finally, given a new sentence in the prediction phase, each CNN in the ensemble learning component will predict the label of the sentence and the label predicted by the most CNNs will be chosen as the final prediction result.

6 Experiment & Results

In this section, we evaluate the performance of our approach for automatically identifying intention categories. We first present the experiment design, including the evaluation setting and research questions. Then we present our experiment results.

6.1 Experiment Design

6.1.1 Evaluation Settings

The experimental environment is a desktop computer equipped with Nvidia GTX 1080 GPU, Intel(R) Core(TM) i7-6700 CPU and 16GB RAM, running Ubuntu 16.04 LTS. We evaluate the performance of our approach in the following three settings:

10-Fold-Cross-Validation Setting. We first evaluate our approach using Di Sorbo et al.'s published dataset (i.e., email data). Di Sorbo et al. studied two projects (i.e., Ubuntu and Qt), and looked at both projects to manually summarize linguistic patterns. To be fair, our approach also needs to be allowed to look into both projects to *automatically* learn models. Thus, we evaluate our approach using 10 times 10-fold-cross-validation [23]. Specifically, we first randomly shuffle the dataset and divide it into ten folds of approximately equal size by a stratified random sampling. Then, each fold is used as a testing dataset to evaluate the prediction model built on the other nine folds (i.e., training dataset). The entire process is repeated ten times to alleviate possible sampling bias in random shuffle and sampling, and we record the average evaluation results.

Cross-Project Setting. We then evaluate the performance of our approach using the 4 projects in our dataset considering cross-project prediction. Each time, we choose one project as "target project" (i.e., for prediction), then we refer to the other three projects as "source projects". We build a prediction model by using all sentences in source projects, and predict the labels of all sentences in target project. The reason is that we would like to investigate whether our prediction model built on source projects can be generalized to other projects. As shown in Section 3, the annotation effort is large, and it is impractical to annotate sufficient data within every new project that we want to apply our proposed approach to build a prediction model. Finally, since our approach introduces randomness (e.g., the parameters of each filter in CNN are randomly initialized), we repeat the cross-project prediction 10-times and record the average evaluation results.

Cross-Discussion-Type Setting. We have annotated sentences from two different kinds of developer discussions, i.e., issue reports (our dataset), and emails (Di Sorbo et al.'s dataset). In this setting, we evaluate the performance of our approach by training a prediction model using all sentences in Di Sorbo et al.'s (our) dataset to predict sentences in our (Di Sorbo et al.'s) dataset. Due to randomness involved in our approach, we also repeat the cross-discussion-type prediction 10-times and record the average evaluation results.

Note that when we compare our approach with Di Sorbo et al.'s patterns, we remove sentences belonging to *aspect evaluation* and *meaningless* in our dataset since these two intentions are not part of Di Sorbo et al.'s taxonomy and covered by their linguistic patterns. Also, since we merge *opinion asking* into *information seeking*, if a sentence is classified by Di Sorbo et al.'s patterns as *opinion asking*, we consider it as *information seeking*.

Evaluation Metrics. Followed by Di Sorbo et al.'s study [10], we use accuracy, precision, recall, and F1-score to evaluate the performance of our approach. Accuracy is the proportion of sentences that are correctly classified among all sentences for all classes. Precision for class C_i is the proportion of sentences that are correctly classified as class C_i among all sentences that are classified as class C_i. Recall for class C_i is the proportion of sentences that are correctly classified as class C_i among all sentences that belong to class C_i. F1-score for class C_i is the harmonic mean of its precision and recall.

Accuracy evaluates the overall performance of an approach, while precision, recall and F1-score evaluate the performance of an approach for a specific intention category. These evaluation metrics are widely used in previous studies of software engineering studies that involve classification process [73]–[77].

6.1.2 Research Questions

In this paper, we investigate the following four research questions:

RQ1: How effective is our approach based on deep learning?

Our approach aims to classify sentences of developer discussions from issue tracking systems. For the approach to be useful, we need to consider how accurate it is in sentence classification and how it compares with existing approaches. To answer this research question, we investigate three specific sub-questions, as shown below:

RQ1-1: How does our approach compare to Di Sorbo et al.'s approach?

Since our taxonomy of user intentions is an extension of Di Sorbo et al.'s work, we choose their linguistic pattern-based approach as a baseline and compare it with our approach. We compare the two approaches using the three experiment settings described in Section 6.1.1.

RQ1-2: How does our approach compare to other automated approaches in related studies?

There are other approaches that also classify sentences into categories, other than intentions. Thus, we investigate how the approaches compare in performance when adapted to our task (i.e., mining intentions of sentences). Specifically, we compare our approach with the following three approaches:

CNN: In Section 5.2 we introduced how we adapt Kim's CNN architecture to our work. We use the implementation of their approach based on TensorFlow. The code is published in GitHub [68]. Note that Kim's CNN did not contain the batch normalization layer and we use the default hyperparameter setting in the published source code.

NLP: Gu et al. proposed a classification technique in which they designed text features based on natural language processing (NLP) to distinguish five categories of Android App review sentences [22]. Their approach can be adapted to solve our problem, since all the features extracted by their approach exist for any types of sentences. Specifically, their approach requires Stanford CoreNLP tools [55] to parse a sentence. We re-implement their approach using the same tool and compare it with our approach.

SMO, LibSVM, NBM, RF and kNN: In general, the intention mining task belongs to text classification. Many traditional text classification techniques are based on Vector Space Model (VSM) using bag-of-words features. Each unique word is regarded as a feature, and a sentence is represented as a vector of feature values. In practice, bag-of-words features extracted from raw text data requires basic preprocessing. In this paper, we follow previous studies [34], [78] to do three basic pre-processing steps, including TF-IDF transformation and stemming and stop words removal. Since there are many classification techniques that can be applied to VSM-based data representation, we choose 4 different classification techniques to build different classifiers, namely Support Vector Machine (SVM), Naive Bayes Multinomial (NBM), Random Forest (RF), and k-Nearest Neighbor (kNN). These classification techniques are also widely used in previous studies [23], [74], [79]. Our implementation of these classifiers is based on Weka [80], which is a suite of machine learning software written in Java. Specifically, Weka provides two different implementations for SVM, namely SMO [81] and LibSVM [82]. We evaluate both SMO and LibSVM in our experiment. All of these classifiers are evaluated with the default hyper-parameter setting in Weka.

DE-SVM and Auto-Weka: Recently, a number of studies have shown that deep learning could be outperformed by traditional machine learning approaches when they are trained properly. For example, Fu and Menzies [24] proposed DE-SVM, which applied a simple optimizer called differential evolution to carefully tune SVM, and they found that DE-SVM outperforms a CNN based approach for binary classification. Another example is Fakhoury et al.'s work [25], in which they applied Auto-Weka [83] to automatically search for the optimal classifier and corresponding hyperparameter settings to maximize the performance for the task of linguistic smell detection. The optimization algorithm of Auto-Weka is based on Bayesian optimization, and it was reported to perform better than a CNN-based approach. Thus, in this paper, we also follow these two studies to apply hyperparameter tuning algorithms for different machine learning models. Specifically, DE-SVM only focuses on tuning for LibSVM, and we follow Fu and Menzies to use differential evaluation to tune the same hyperparameters for LibSVM, including *kernel* ('liner', 'poly', 'rbf' or 'sigmoid'), *C* (1 to 50), *gamma* (0 to 1) and *coef0* (0 to 1). For Auto-Weka, we directly integrate its API with our Java code and it would automatically choose the best classifier with tuned hyperparameters. The objectives of both DE-SVM and Auto-Weka are to maximize accuracy.

RQ1-3: How much time and memory does it take for our approach to run?

If our approach cannot run efficiently in a reasonable cost of time and memory, developers might not be willing to use it in practice even if it could achieve a high accuracy. To improve the training speed of our approach, we integrate batch normalization with CNN. Thus, we are interested to investigate its benefit on time efficiency. On the other hand, our approach uses GPU acceleration, which requires to load the network into the memory of GPU. In general, setting a larger value for certain hyperparameters like the dimension size of word embedding or the number of filters would increase the memory cost, since more neurons would be created for the network. Thus, we are interested to investigate the memory cost of our approach. To answer this research question, we first evaluate the training time cost of our approach with and without BN, and other baseline approaches under different experiment settings. Then we evaluate the memory cost of our approach and compare it with other baseline approaches.

RQ2: Does our automatic hyperparameter tuning approach find the best setting of hyperparameters?

Since our automatic hyperparameter tuning runs in a greedy way, the full search space is pruned a lot. Specifically, our approach only enumerates $4 + 4 + 10 = 18$ different settings since the other hyperparameters are fixed when we enumerate one specific hyperparameter. If we enumerate all possible combinations of the three hyperparameters (i.e., using grid search), there would be $4 * 4 * 10 = 160$ different settings. Thus, we investigate whether our automatic hyperparameter tuning approach finds a setting whose performance is close to that using the best hyperparameter setting. To answer this research question, we first use grid search to enumerate all possible settings of hyperparameters and choose the setting that achieves the best accuracy to compare with our approach. Note that during grid search, the search spaces of both the dimension size of word embedding and the number of filters are still from 128 to 320 with a step of 64, and the candidate combinations of filter heights are the same with those presented in Section 5.3. Then we compare the hyperparameters selected and the corresponding accuracy results achieved by our approach and grid search.

RQ3: Do developers agree with our taxonomy of intentions and the classification results of our approach?

One of our initial questions to investigate in this paper is whether Di Sorbo et al.'s intention mining approach can be generalized across a different kind of developer discussion and a different set of projects. Our preliminary exploration during the manual labeling process has shown that refinements were needed to the intention categories and that many sentences were not covered by the linguistic patterns manually learned by Di Sorbo et al.'s work. Although we refined the taxonomy of intentions for developer discussions from issue tracking systems and we manually labelled more sentences to evaluate our automated approach, it is still unknown whether developers agree with our taxonomy of intentions and the classification results of our approach. As a further investigation into this generalization question, we conduct a user study in which we ask professional developers to assess whether they agree with intention categories produced using our approach that was trained with data manually annotated by the authors on different projects. The details of the user study are presented in Section 7.

RQ4: Can our approach help improve a downstream automated software engineering task?

The output of our approach is a set of classified sentences with different intentions. However, it is still not clear whether these classified sentences can be applied in software engineering tasks. To further demonstrate the value of our proposed approach, we apply it for a specific task, namely rectifying misclassified issue reports. The details of this application are presented in Section 8.

TABLE 3
Comparison of our approach with Di Sorbo et al.'s approach in terms of accuracy

Approach	Issue	Email	Issue to Email	Email to Issue
Ours	**0.839**	**0.791**	0.658	**0.579**
Pattern	0.305	0.746	**0.746**	0.305

TABLE 4
Comparison of our approach with Di Sorbo et al.'s approach in terms of F1-score for each intention category

Setting	App.	Intention Category					Avg.
		IG	IS	FR	SP	PD	
Issue	Ours	0.802	0.904	0.793	0.788	0.818	**0.821**
	Pattern	0.294	0.511	0.420	0.283	0.600	0.422
Email	Ours	0.782	0.883	0.792	0.742	0.887	**0.817**
	Pattern	0.743	0.874	0.789	0.733	0.879	0.804
Issue to Email	Ours	0.563	0.809	0.578	0.443	0.760	0.631
	Pattern	0.743	0.874	0.789	0.733	0.879	**0.804**
Email to Issue	Ours	0.488	0.678	0.580	0.428	0.520	**0.538**
	Pattern	0.294	0.511	0.420	0.283	0.600	0.422

6.2 Experiment Results

RQ1-1: How does our approach compare to Di Sorbo et al.'s approach?

Table 3 presents the accuracy achieved by our approach and Di Sorbo et al.'s approach under different experiment settings. Table 4 - 6 present the F1-score, precision and recall achieved by the two approaches for each intention category under different experiment settings, respectively. The name of each intention category is represented by its corresponding acronym, and the best result for each evaluation setting is highlighted in bold.

10-Fold-Cross-Validation Setting. In summary, our approach achieves an average accuracy of 0.791 on the email dataset, while Di Sorbo et al.'s approach achieves an accuracy of 0.746. The results show that our approach is able to learn the semantic patterns of sentences in Di Sorbo et al.'s dataset and performs slightly better than Di Sorbo et al.'s manually summarized patterns. Most importantly, our approach saves the manual effort required to summarize the linguistic patterns.

Cross-Project Setting. On average across the issue reports from four projects, our approach achieves accuracy of 0.839, which improves the pattern-based approach (average accuracy of 0.305) by 175%. Since a large number of sentences cannot be covered by any one of Di Sorbo et al.'s linguistic patterns, it is reasonable that their approach achieves low accuracy.

Cross-Discussion-Type Setting. When using our issue reports as training data, our approach achieves an accuracy of 0.658 when predicting sentences in Di Sorbo et al.'s email dataset, which is relatively lower than the accuracy achieved by their approach (i.e., 0.746). This is reasonable since their linguistic patterns are manually created based on sentences in their dataset, thus fitting well to these sentences. When using Di Sorbo et al.'s dataset as training data, our approach achieves an average accuracy of 0.579 across the four projects in our issue report dataset, which improves Di Sorbo et al.'s approach by 90%. This result indicates

TABLE 5
Comparison of our approach with Di Sorbo et al.'s approach in terms of precision for each intention category

Setting	App.	Intention Category					Avg.
		IG	IS	FR	SP	PD	
Issue	Our	0.752	0.909	0.815	0.809	0.858	**0.829**
	Pattern	0.514	0.947	0.552	0.477	0.828	0.664
Email	Our	0.733	0.888	0.814	0.762	0.930	0.825
	Pattern	0.884	0.981	0.748	0.910	0.958	**0.896**
Issue to Email	Our	0.428	0.831	0.645	0.654	0.803	0.672
	Pattern	0.884	0.981	0.748	0.910	0.958	**0.896**
Email to Issue	Our	0.705	0.663	0.460	0.383	0.502	0.543
	Pattern	0.514	0.947	0.552	0.477	0.828	**0.664**

TABLE 6
Comparison of our approach with Di Sorbo et al.'s approach in terms of recall for each intention category

Setting	App.	Intention Category					Avg.
		IG	IS	FR	SP	PD	
Issue	Our	0.867	0.900	0.774	0.778	0.786	**0.821**
	Pattern	0.205	0.350	0.339	0.201	0.472	0.313
Email	Our	0.845	0.879	0.773	0.733	0.853	**0.816**
	Pattern	0.641	0.788	0.835	0.613	0.812	0.738
Issue to Email	Our	0.820	0.787	0.525	0.335	0.722	0.638
	Pattern	0.641	0.788	0.835	0.613	0.812	**0.738**
Email to Issue	Our	0.372	0.694	0.781	0.484	0.539	**0.574**
	Pattern	0.205	0.350	0.339	0.201	0.472	0.313

that while the majority of Di Sorbo et al.'s taxonomy of intentions are still effective for developer discussions in issue tracking systems, a refinement is needed for their linguistic patterns to cover more sentences in this kind of developer discussions.

Qualitative Analysis. To gain more insights about the advantages and disadvantages of our approach and Di Sorbo et al.'s approach, we conduct a qualitative analysis, in which we manually check the sentences that are misclassified by any one of the two approaches. For Di Sorbo et al.'s approach, the results show that it achieves a much lower accuracy on the issue dataset than that achieved on the email dataset. Specifically, we note that the recall of all categories achieved by Di Sorbo et al.'s approach on the issue dataset are quite low (i.e., less than 0.472). One reason is that a lot of sentences in the issue dataset are not covered by Di Sorbo et al.'s linguistic patterns. However, we also note the precision of *information giving*, *feature request* and *solution proposal* achieved by Di Sorbo et al.'s approach are also relatively low (i.e., less than 0.552). By manually checking the misclassified sentences, we find that although some sentences contain the linguistic patterns, their true intentions can be completely different. For example, the sentence "*I should add, the threading point is partly my speculation based on what I observed in performance running xgboost and other applications that use a multithreaded blas through R*" in TensorFlow Issue #491 is misclassified as *solution proposal*, while it should belong to *information giving*. This sentence is misclassified because it contains the linguistic pattern "*[someone] should add [something]*", however, the "thing" that is added here is not a feature but a threading point to provide more details about the developer's observation. Another example is the sentence "*An easy way to reproduce it is to run bazel fetch //tensorflow/contrib/session_bundle/.*" in TensorFlow Issue #4312, which is misclassified as *solution proposal*, while it should belong to *information giving*. This sentence contains the linguistic pattern "a way is", however, its intention is to provide the command to reproduce the bug. From these examples, we can see that since most of the linguistic patterns are just some key phrases along with some placeholders indicating generic subjects (i.e., somebody) or generic direct objects (i.e., something), at times it is difficult for Di Sorbo et al.'s approach to capture the high-level intention of the whole sentence.

On the other hand, we also investigate why our approach performs worse under the cross-discussion-type setting (i.e., Issue-to-Email and Email-to-Issue), when compared with the results achieved by our approach using the email dataset or issue dataset alone. For example, under the Issue-to-Email setting, our approach only correctly identified about 33% of all sentences belonging to solution proposal. One reason is that many sentences of solution proposal in Di Sorbo et al.'s email dataset are a bit vague to understand their intentions. For example, the sentence "*One way would be to add them in an #ifdef Q_QDOC block and document them.*" in QT mailing list (2014-August/017822) is labelled as solution proposal, while our approach classifies it as feature request. The email that contains this sentence is about how to deal with some special member functions when QT framework has moved to C++11. In general, this is a discussion about the implementation of C++11 rules, and both intentions (i.e., solution proposal and feature request) should be reasonable under such context. This example also shows that, in some cases, the proposed solutions can also be adding new features. Thus, while our approach misclassified more sentences under the Issue-to-Email setting, some of the misclassified results are still reasonable in practice. As for the Email-to-Issue, our approach performs even worse. An additional reason is that we only have no more than 1,000 sentences as training dataset, which makes it difficult to capture enough patterns of different intentions.

RQ1-2: How does our approach compare to other automated approaches in related studies?

Cross-Project Setting. Table 7 presents the accuracy results achieved by each approach, and the best result for each project is highlighted in bold. The accuracy achieved by our approach ranges between 0.705 and 0.867, with an average of 0.816. In comparison, the average accuracy achieved by CNN, NLP, SMO, LibSVM, NBM, RF, kNN, DE-SVM and Auto-Weka are 0.711, 0.593, 0.533, 0.247, 0.441, 0.420, 0.245, 0.539 and 0.458, respectively. In summary, our approach improves the average accuracy over CNN, NLP, SMO, LibSVM, NBM, RF, kNN, DE-SVM and Auto-Weka by 15%, 38%, 53%, 230%, 85%, 94%, 233%, 51% and 78% respectively.

TABLE 7
The accuracy achieved by different approaches for each project in our dataset

Approach	TensorFlow	Bootstrap	Docker	VScode	Average
Ours	**0.705**	**0.828**	**0.867**	**0.862**	**0.816**
CNN	0.589	0.743	0.766	0.744	0.711
NLP	0.519	0.601	0.641	0.611	0.593
SMO	0.457	0.570	0.565	0.539	0.533
LibSVM	0.240	0.272	0.236	0.239	0.247
NBM	0.402	0.423	0.498	0.442	0.441
RF	0.352	0.430	0.463	0.435	0.420
kNN	0.214	0.241	0.250	0.273	0.245
DE-SVM	0.469	0.581	0.559	0.545	0.539
Auto-Weka	0.425	0.487	0.469	0.450	0.458

TABLE 8
The average F1-score achieved by different approaches across all four projects for each intention category in our dataset

Approach	IG	IS	FR	SP	PD	AE	ML	Avg.
Ours	**0.781**	**0.911**	**0.781**	**0.782**	**0.804**	**0.796**	**0.860**	**0.816**
CNN	0.670	0.850	0.690	0.579	0.674	0.698	0.796	0.708
NLP	0.597	0.614	0.610	0.486	0.609	0.545	0.662	0.589
SMO	0.504	0.495	0.571	0.535	0.690	0.505	0.587	0.555
LibSVM	0.395	0.000	0.000	0.000	0.000	0.000	0.000	0.056
NBM	0.437	0.423	0.403	0.374	0.521	0.486	0.525	0.452
RF	0.361	0.421	0.439	0.227	0.635	0.374	0.452	0.416
kNN	0.065	0.300	0.072	NaN	0.480	0.093	0.223	0.206
DE-SVM	0.535	0.461	0.559	0.528	0.662	0.527	0.545	0.545
Auto-Weka	0.492	0.192	0.507	0.533	0.677	0.309	0.236	0.421

TABLE 9
The average precision achieved by different approaches across all four projects for each intention category in our dataset

Approach	IG	IS	FR	SP	PD	AE	ML	Avg.
Ours	**0.732**	**0.916**	**0.803**	**0.803**	**0.843**	**0.804**	**0.934**	**0.834**
CNN	0.651	0.864	0.705	0.614	0.654	0.719	0.798	0.715
NLP	0.610	0.602	0.609	0.552	0.639	0.609	0.539	0.594
SMO	0.536	0.414	0.671	0.633	0.714	0.618	0.493	0.583
LibSVM	0.247	0.000	0.000	0.000	0.000	0.000	0.000	0.035
NBM	0.442	0.440	0.392	0.388	0.509	0.472	0.610	0.465
RF	0.514	0.302	0.735	0.565	0.738	0.590	0.321	0.538
kNN	0.438	0.227	0.646	0.319	0.666	0.414	0.130	0.406
DE-SVM	0.449	0.468	0.737	0.662	0.700	0.581	0.657	0.608
Auto-Weka	0.350	0.556	0.633	0.570	0.680	0.513	0.739	0.577

TABLE 10
The average recall achieved by different approaches across all four projects for each intention category in our dataset

Approach	IG	IS	FR	SP	PD	AE	ML	Avg.
Ours	0.844	**0.907**	**0.762**	**0.772**	**0.773**	**0.788**	0.800	**0.807**
CNN	0.700	0.837	0.678	0.582	0.704	0.684	0.798	0.712
NLP	0.599	0.633	0.615	0.459	0.596	0.497	**0.865**	0.609
SMO	0.483	0.623	0.510	0.470	0.674	0.436	0.738	0.562
LibSVM	**1.000**	0.000	0.000	0.000	0.000	0.000	0.000	0.143
NBM	0.444	0.409	0.424	0.383	0.538	0.521	0.467	0.455
RF	0.283	0.700	0.329	0.144	0.564	0.276	0.788	0.441
kNN	0.035	0.447	0.038	0.029	0.388	0.054	0.820	0.259
DE-SVM	0.676	0.462	0.459	0.445	0.635	0.509	0.469	0.522
Auto-Weka	0.832	0.116	0.444	0.521	0.681	0.232	0.142	0.424

We also investigate the performance of each approach for each intention category. Table 8 - 10 presents the F1-score, precision and recall of each category achieved by each approach, respectively. Due to space limitation, the name of each category is abbreviated and we report the average results of each category across the four projects. For each intention category, the best results of F1-score are highlighted in bold. Our approach achieves the best performance for all categories. Additionally, the F1-score achieved by our approach for each category is higher than 0.78, which is higher than that achieved by other approaches. As for precision and recall, our approach achieves the best results for most categories, while LibSVM achieves the best recall for *information giving* and NLP achieves the best recall for *meaningless*. Note that LibSVM achieves a recall of 1.0 for *information giving*, but it achieves a recall of zero for all the other categories. This indicates that LibSVM with the default parameter setting classifies all sentences as *information giving*. Specifically, LibSVM uses the *rbf* kernel by default, while DE-SVM also chooses *rbf* kernel as the best kernel in several cases. However, with carefully tuned values of hyperparameters like *gamma* and *coef0*, DE-SVM substantially improves the performance of LibSVM. The result is also consistent with Fu and Menzies' finding that when applying LibSVM for text classification, a carefully tuning is necessary [24]. On the other hand, Auto-Weka does not outperform DE-SVM in our experiment. Specifically, Auto-Weka chooses RF (i.e., random forest) as the best classifier in most cases and this tuned RF classifier does outperform its default version. Finally, we also note that SMO seems to be the best classifier among all the default machine learning models, however, both DE-SVM and Auto-Weka didn't choose SMO as the base model for hyperparameter tuning. Thus, we apply a simple grid search algorithm to tune SMO. We find that when setting its kernel as *NormalizedPolyKernel* and *exponent* as 1.3, we can achieve an average accuracy of 0.561, which improves its default version by 5%. However, this tuned SMO still cannot outperform our approach.

Other Settings. Table 11 presents the accuracy results achieved by each approach under 10 times 10-fold-cross-validation using Di Sorbo et al.'s dataset (second row) and cross-discussion-type setting using both datasets (the last two rows). We observe that our approach improves other approaches by a substantial margin.

Qualitative Analysis. Since all the evaluation results show that our approach improves the baseline approaches in terms of accuracy by a substantial margin, we are interested in those sentences that are correctly classified by our approach only. We manually checked a number of cases and found that both our approach and baseline approaches can easily classify sentences with important keywords (e.g., *crash* and *error*) in sentences belonging to *problem discovery*. However, our approach can also correctly classify sentences that are relatively vague. For example, the sentence "*I set the breakpoint as indicated in #8859 (comment) but this code wasn't called when clicking on the 'Install' button of the new extension panel.*" does not contain any words like *crash* or *error*, but it belongs to *problem discovery* since the user describes an unexpected behavior. Our approach successfully identified this sentence and we hypothesize that our approach has learned the hidden linguistic pattern during training.

Finally, we also analyze the sentences misclassified by our approach. We find that some of these sentences contain relatively complicated subordinate clause. For example, the sentence "*I also have a feeling that using .{size}-{n} as a class instead of .col-{size}-{n} might cause some problems down the line.*" in Bootstrap Issue #17228 belongs to *aspect evaluation* while our approach classifies it as *problem discovery*. We hypothesize that our approach may have overweighted "cause some problems" phrase and ignored "have a feeling that", thus leading to misclassification. To solve this problem, more sentences with subordinate clause are likely needed for training.

RQ1-3: How much time and memory does it take for our approach to run?

Time Cost. Table 12 shows the training time cost of each approach. The training time cost for Di Sorbo et al.'s approach is not presented, since they did not show the time cost to manually summarize the linguistic patterns. The results show that, when using BN, our approach requires about half an hour to train the CNN using our dataset. We believe the training time cost is acceptable, since the prediction model only needs to be trained once and our approach achieves a high accuracy. If BN is not used, our

TABLE 11
The accuracy achieved by different approaches using the email and issue report dataset.

Approaches	Email	Email to Issue	Issue to Email
Ours	**0.791**	**0.579**	**0.658**
CNN	0.710	0.522	0.569
NLP	0.575	0.460	0.483
SMO	0.707	0.370	0.450
LibSVM	0.273	0.231	0.173
NBM	0.607	0.349	0.347
RF	0.693	0.329	0.383
kNN	0.604	0.239	0.370
DE-SVM	0.714	0.374	0.402
Auto-Weka	0.707	0.349	0.329

TABLE 12
Training time cost of each approach under different experiment setting

Approach	Issue	Email	Issue to Email	Email to Issue
Our (with BN)	30min	6min	38min	7min
Our (without BN)	5.6h	1h	7.2h	1.2h
Kim's CNN	20min	4min	24min	5min
NLP	4min	1min	6min	1min
SMO	11s	2s	12s	3s
LibSVM	9s	1s	10s	2s
NBM	1s	1s	1s	1s
RF	3min	1min	4min	1min
kNN	1s	1s	1s	1s
DE-SVM	20min	10min	16min	11min
Auto-Weka	16min	15min	15min	15min

approach would require several hours to train. In summary, batch normalization accelerates the training speed of CNN by at least 10 times. As for the other approaches, most prediction models based on bag-of-words features achieve the fastest training speed. However, they cannot achieve a high accuracy. The NLP-based approach requires about 6 minutes to train from 5K sentences. Finally, our approach only needs several seconds to classify thousands of sentences in the prediction phase, which indicates that our approach is efficient enough to be used in practice.

Memory Cost. Our approach runs on a Nvidia GTX 1080 GPU. This GPU has 8GB memory available. During our automatic hyperparameters tuning process, we enumerate the dimension size of word embedding and the number of filters from 64 to 320 with a step of 64. We cannot increase these two hyperparameters further beyond 320 since the GPU's memory would not be sufficient in such a case. As a comparison, all the other baseline approaches run on a CPU and their memory cost is rather low (i.e., no more than 2G). Thus, the high cost of memory is a drawback of our approach. To solve the memory issue, we can use a better GPU or multiple GPUs since the current GPU (i.e., Nvidia GTX 1080) used in our paper is originally designed for PC gamers and its official selling price is about 500 dollars in 2018. In future work, we also plan to improve the architecture of our network to reduce its size. Note that increasing the amount of training data would not increase the size of the network; this is the case since we use Mini-batch Gradient Descent [84] to train the network, which means that we only feed a fixed number of sentences into the network for each iteration of the training process.

RQ2: Does our automatic hyperparameter tuning approach find the best setting of hyperparameters?

Since our automatic hyperparameter tuning runs in a greedy way, the full search space is pruned a lot. Specifically, our approach only enumerates $4+4+10=18$ different settings since the other hyperparameters are fixed when we enumerate one specific hyperparameter. If we enumerate all possible combinations of the three hyperparameters (i.e., using grid search), there would be $4*4*10=160$ different settings. In this RQ, we investigate whether our automatic hyperparameter tuning approach finds a setting whose performance is close to that using the best hyperparameter setting. To answer this research question, we first use grid search to enumerate all possible settings of hyperparameters and choose the setting that achieves the best accuracy to compare with our approach. Note that during grid search, the search spaces of both the dimension size of word embedding and the number of filters are still from 128 to 320 with a step of 64, and the candidate combinations of filter heights are the same with those presented in Section 5.3. Then we compare the hyperparameters selected and the corresponding accuracy results achieved by our approach and grid search.

Table 13 presents the hyperparameters selected and the corresponding accuracy results achieved by our approach and grid search. Although our approach does not select the same hyperparameters for three out of the four projects, the accuracy achieved by our approach is slightly better than that achieved by grid search. This shows that ensemble learning can help improving the performance. We also notice that, both the dimension size and number of filters selected by our approach or grid search are larger than Yoon Kim's default setting, which indicates that hyperparameter tuning is needed when applying CNN in different tasks with dataset collected from different domains.

7 User Study

One of our initial questions to investigate in this paper is whether Di Sorbo et al.'s intention mining approach can be generalized across a different kind of developer discussion and a different set of projects. We showed that refinements were needed to the intention categories and that an approach that could learn patterns to identify categories, instead of patterns being learned manually as in Di Sorbo et al.'s approach, can identify intention categories in a wider set of projects with higher accuracy. However, the question still remains with the approach we introduce whether it generalizes in the sense of finding intention categories accurately across more projects.

As a further investigation into this generalization question, we conducted a user study in which we asked professional developers to assess whether they agree with intention categories produced using our approach that was trained with data manually annotated by the authors on different projects. For this study, we trained a prediction model using all of the 5,408 sentences in the dataset we created and which we described in Section 3. We then selected another four popular projects in GitHub: Three.js, Ruby on Rails, OpenCV and Scikit-Learn. For each project, we randomly selected 20 issues and randomly extracted 500 sentences from these issues. We then applied our approach (prediction model) to classify each of the 500 sentences into an intention category. Finally, we randomly selected 5 sentences from each category forming a dataset for the user

TABLE 13
Comparison of the hyperparameter settings and corresponding accuracy achieved by grid search and our automatic tuning approach

Hyperparameters	TensorFlow		Bootstrap		Docker		VS Code	
	Grid Search	Auto Tuning	Grid Search	Auto Tuning	Grid Search	Auto Tuning	Grid Search	Auto Tuning
Dimension size	256	320	256	256	320	256	320	320
Number of filters	192	256	256	256	256	192	192	320
Combination of filter heights	(1,2,3,4,5,6)	Ensemble	(1,2,3,4,5)	Ensemble	(2,3,4,5,6)	Ensemble	(3,4,5,6)	Ensemble
Accuracy	0.701	0.705	0.817	0.828	0.865	0.867	0.852	0.862

study consisting of 20 sentences for each intention category and 140 sentences in total.

We recruited 11 developers from an outsourcing company which has more than 2,000 employees and mainly does outsourcing projects for US and European corporations (e.g., StateStreet Bank, Cisco, and Reuters). Each of the recruited developers has previous experience as a developer in IT companies or as a contributor in open source software projects.

Table 14 presents the demographics of the 11 developers, including their working experience counted in years, the programming language mostly used in their daily work, their job role, their experience in each evaluated project (the name of each project is abbreviated). Specifically, for each project, '-' means the developer does not have any experience in this project; '+' means the developer has necessary domain knowledge or has experience in similar projects; '++' means the developer has rich experience in the project itself. According to the demographics, the 11 developers are of different professional experience, varying from 2 years to 8 years, with an average professional experience of 4.2 years. This diversity of experience helps to reduce any bias where agreement with predictions is only achieved by a specific group of developers (e.g., novice or senior developers).

We explained our taxonomy of intentions to the developers and showed them some example sentences for each category. For the user study dataset, we asked each developer to read each sentence (140 per developer) and indicate whether the predicted intention category by our approach is correct. We also provided the developers with the context of the sentences, in the form of corresponding paragraphs and issue numbers as we did for the creation of the training dataset (Section 3). The developers were aware that it was an approach we developed that was producing the intention categories. Although the recruited developers do not have rich experience with every project from which the sentences in the user study dataset were drawn, we believe their experience in development allow them to identify the intentions of the sentences.

Table 15 presents the percentage of sentences for each category that were agreed upon by the developers. We present the results in two different settings, where *Majority* means a sentence is considered as correctly classified if the majority of developers agree on it (i.e., when at least 6 developers agree), and *All* means a sentence is considered as correctly classified if all developers agree on it. The results show that, on average across the seven categories, about 89% of the classification results are agreed by the majority of developers, and 73% of the classification results are agreed by every developer.

For some classified sentences that are not agreed upon by developers, we also received some feedback from developers about their own opinions on what should be the intention category of the sentence. Delving further into these cases, we found two major situations:

TABLE 14
Demographics of the 11 developers

Dev.	Exp.	Language	Job Role	Experience in Projects			
				Th	Ru	CV	Sci
P1	2 years	Java	QA	-	+	-	-
P2	4 years	Java\Python	QA	-	+	-	+
P3	6 years	Python\Go	DevOps	-	+	+	+
P4	2 years	Java\SQL	Database	-	-	-	-
P5	8 years	JavaScript	Web	+	++	+	+
P6	3 years	JavaScript	Web	+	+	-	-
P7	5 years	Java	Web	+	+	-	-
P8	5 years	Java\Python	Data Mining	-	-	+	++
P9	4 years	C++	Mobile App	+	-	-	-
P10	4 years	Java	Mobile App	-	+	-	-
P11	3 years	C++	Network	-	+	-	-

TABLE 15
User study results

Agreement	IG	IS	FR	SP	PD	AE	ML	Avg
Majority	0.85	0.85	0.80	0.90	0.90	0.90	1.00	0.89
All	0.60	0.65	0.75	0.60	0.75	0.80	0.90	0.73

1) The sentence contains multiple intentions. For example, the sentence *"And why do stitching modules crashes even with CUDA explicity disabled?"* in OpenCV Issue #7438 is classified as *problem discovery* since it reports an unexpected crash. However, many developers also regard it as *information seeking* since the user is asking why.
2) The classifier does not consider context information. For example, the sentence *"I believe, the main point here is to have some standard, 'official' container for such data."* in OpenCV Issue #8428 is classified as *aspect evaluation* since the user is expressing his view on the main point. However, one developer in our study who has rich experience in open source community inferred that this sentence should be considered as *feature request*. By checking the corresponding issue, we found that, indeed, it is a request for a new feature, i.e., support for a half-precision floating point which only occupies 16 bits in memory.

To summarize, our user study results show the developers agreed on most of the predicted intention categories of the 140 sentences, thus providing an additional evidence on the generalizability of our approach.

Finally, to understand practitioners' opinions and expectations on intention mining, we conducted a survey with the 11 participants in our user study. Our survey has three questions, as shown below:

- **Q1:** *Do you perceive our taxonomy of intentions in developer discussions as useful/meaningful?*

- **Q2:** *Would you be interested in using a tool performing intention mining for your work? If so, what kind of work would you apply intention mining to?*
- **Q3:** *Could you explain the reasons for your answers in Q1 and Q2? (Optional)*

For Q1, eight participants agreed that our taxonomy is useful/meaningful, and the major reason is that the taxonomy has fully covered the common types of intentions they dealt with in developer discussions. The other three participants did not agree, and one of them gave a reason. This participant stated that although the taxonomy seems to be meaningful, he could not figure out how to use it in his daily work.

For Q2, only six participants showed their interest in applying intention mining to their work, and most of them only gave general reasons, such as "This seems fancy" or "This might help us better understand our project". For the other five participants who showed no interest, their major reasons are that they do not have such requirement or data for analysis. Nevertheless, one participant introduced our work to his team leader, who showed great interest in our work and shared with us a scenario that can benefit from intention mining. This team developed their project for an important customer, who required regular delivery of releases. To better track the development progress, the team leader would like to analyze the developer discussions in the company's internal channels (i.e., discussions through mailing lists or the bug tracker). Especially, he would like to have a tool which can automatically extract specific discussions about bugs and features. Based on this data, textual analysis (e.g., name entity recognition) could be more effectively applied to tell what features or bugs are discussed recently. If developers are found to be frequently discussing features or bugs from early releases, then it might indicate that the project's delivery for the next release is at risk of delay. Thus, intention mining seems promising to help monitor the development progress.

In summary, most participants perceived our taxonomy of intentions in developer discussions as useful/meaningful, however, it is still unclear how intention mining could be applied to practitioners' daily work. Thus, we encourage future research to investigate how to apply intention mining to solve more software engineering tasks.

8 Application on Rectifying Misclassified Issue Reports

In this section, to further demonstrate the value of our proposed approach, we apply it for a specific task, namely rectifying misclassified issue reports.

A number of studies have proposed approaches to mine data from issue tracking systems to predict where bugs will occur in the future (aka. defect prediction [27], [28]). To conduct such study, the first thing is to pick out issue reports that really report bugs. An issue report usually has a tag or a field which records the type of the issue report (e.g., bug report or feature request). The issue reports that are labelled as bug reports are often used as the ground truth data for various mining tasks (e.g., defect prediction). However, Herzig et al. [26] conducted an empirical study and found that many issue reports are misclassified – issue reports labeled as bugs may actually be feature requests and vice versa. This misclassification have been shown to introduce bias and threaten the external validity of studies that build on such data.

TABLE 16
Number of BUG and FR issue reports in each project

Type	HttpClient	Lucene	Tomcat5	Rhino
BUG	298	678	660	299
FR	30	49	32	30

The study of Herzig et al. requires manual examination of a large number of issue reports, which is time-consuming. To help reduce the manual effort in rectifying the misclassified issue reports, we apply our approach to search for sentences that are likely to belong to feature requests in an issue report. If an issue report classified as a bug report (*BUG*) contains n such sentences, we should consider rectifying its label as a feature request (*FR*)[2]. Although there may exist multiple intentions in a comment (i.e., one or more paragraphs) proposed by a developer, we found that in most issue reports of feature requests, the intention of *feature request* usually dominates other intentions. For example, the developer may start with some background description (information giving) or problem statement (problem discovery), and then propose the feature request. On the other hand, it is rare to find sentences belonging to *feature request* in issue reports of bug. Based on these finding, we set $n = 1$ as the default threshold so that we can find more misclassified BUGs.

Based on the dataset published by Herzig et al., we conducted a case study on four open source projects, namely HttpClient, Lucene, Tomcat5 and Rhino, with 2,076 issue reports in total. Among the four projects, HttpClient and Lucene use JIRA tracker, while Tomcat5 and Rhino use Bugzilla tracker. Table 16 shows the number of issue reports that truly belong to *BUG* and the number of issue reports that are rectified by Herzig et al. from *BUG* to *FR* in each project. Since Herzig et al.'s dataset only contains the ID of the issue report (along with the original and rectified labels), we first crawl the text description of each issue report in the dataset and split them into sentences. Since our CNN code is written in Python, to make the whole process fully automatic, here we use the Python package NLTK [85] to split sentences and we use regular expression to filter out source code and stack traces. Then we apply our sentence classifier trained from our own dataset (i.e., the four GitHub projects) to classify each sentence into one intention category. Based on the classified sentences, we design a simple heuristic approach, which considers an issue report as *FR* if it contains at least one sentence belonging to *feature request*.

Considering the dataset is highly imbalanced (i.e., most issues are *BUG*), we use the Area Under the receiver operator characteristic Curve (AUC) [86] of *FR* class to evaluate our approach. AUC is a robust evaluation metric for imbalanced dataset [87], and a higher AUC score indicates that the classifier has a higher probability to rank a randomly

2. A feature request is also known as a request for enhancement, often shortened as *RFE*.

TABLE 17
AUC scores of our approach and baseline approaches for each project

Approach	HttpClient	Lucene	Tomcat5	Rhino	**Average**
Ours	**0.651**	**0.658**	**0.695**	**0.745**	**0.687**
SMO	0.487	0.552	0.513	0.580	0.533
LibSVM	0.500	0.500	0.500	0.500	0.500
NBM	0.545	0.635	0.593	0.600	0.593
RF	0.500	0.500	0.500	0.517	0.504
kNN	0.540	0.614	0.567	0.557	0.570
LR	0.490	0.543	0.537	0.566	0.534
ADTree	0.510	0.528	0.530	0.500	0.517
DE-SVM	0.513	0.545	0.498	0.533	0.522
Auto-Weka	0.515	0.526	0.498	0.580	0.530

chosen *FR* issue higher than a randomly chosen *BUG* issue. We also compare our approach with traditional text classification approaches. Similar to the baseline approaches introduced in RQ2, we investigate four different classification techniques, namely SVM, NBM, RF and kNN. We also investigate two additional classification techniques, namely Logistic Regression (LR) and ADTree, which are used in a previous study by Antoniol et al. [29], where they applied text mining to classify issue reports as bug reports or feature requests. Since all these classifiers requires training data, we perform 10-times 10-fold cross-validation for each project.

Table 17 shows the AUC scores achieved by our approach and baseline approaches for each project. The results show that our approach outperforms the baseline approaches for each project. On average, our approach achieves an AUC score of 0.69, which improves the best baseline approach (i.e., NBM) by 16%. We also apply the Wilcoxon signed-rank test [88] at 95% significance level and compute the Cliff's delta (δ) [89] when comparing our approach with each baseline. The results show that our approach statistically significantly outperforms (i.e., p-value < 0.05) each baseline with a large improvement (i.e., $|\delta| \geq 0.474$). Another advantage of our approach is that it does not require training data for these projects, which eliminate the effort to label some data for each project, and make it applicable for new projects.

Finally, we manually read the issue reports that are classified as *BUG* by Herzig et al., while our approach classified them as *FR*. We find that some of these issues should belong to *FR* instead. For example, in an issue report of LUCENE[3] that is labelled with *BUG*, our approach identifies a sentence belonging to feature request, which says: *It would be nice if I had an API that would allow me to say "I only want one segment and I want its name to be foo."*. It is clear that the issue reporter is requesting for an enhancement of the Lucene API. Through this case, we can see that our approach can not only identify issues reports of *FR*, but also tells why the issue reports belong to *FR*. As stated by Herzig et al., they cannot guarantee that their manual inspection does not contain errors. Thus, we believe our approach can serve as a support tool to save manual effort and reduce bias when manual inspecting issue reports and rectifying the misclassified ones.

3. https://issues.apache.org/jira/browse/LUCENE-523

9 Discussion

9.1 Interpretability of CNN's classification result

Although our approach outperforms Di Sorbo et al.'s approach in terms of accuracy when classifying sentences of developer discussions from issue tracking systems, one drawback of our approach is that it cannot interpret the classification result. For example, given the sentence "*Therefore, it would be nice if AdaDelta could be added to the set of available optimizers.*" in TensorFlow Issue #516, Di Sorbo et al.'s approach could successfully identify it as *feature request* since it matches the linguistic pattern "*[something] should/could be [verb]*". Thus, the phrase "*AdaDelta could be added*" can be highlighted to interpret why this sentence belongs to the *feature request* category. Such interpretability is important in practice, as it can help to support further automated information extraction processes. For example, Zhou et al. [1] leveraged the specific patterns detected in sentences contained in API documents to translate such sentences to First Order Logic formulae and to detect inconsistencies between source code and API documents. To investigate the interpretability of our approach, we conduct an explorative study, in which we try to locate important patterns in the sentence based on the intermediate outputs of neural network.

When we feed a sentence into the trained CNN model, suppose the network has N filters in total, we would get a high-level feature vector V with N values where each value corresponds to the pooling result of each filter. This feature vector V is fully connected with the output layer Y. Suppose the sentence is classified as a feature request, which corresponds to the i_{th} neuron in the output layer whose output value y_i is calculated as:

$$y_i = b_i + \sum_{j=1}^{N} w_{ij} v_j \tag{7}$$

In this equation, v_j represents the pooling result of the j_{th} filter, while w_{ij} and b_i are weight values automatically learned during training. Since a larger value of y_i indicates that the sentence is more likely to belong to the i_{th} class (i.e., feature request in this example), we can use $w_{ij}v_j$ to evaluate the "contribution" of each filter to the final classification result. In general, a larger value of $w_{ij}v_j$ corresponds to more contribution. Thus, we can pick out the filter whose corresponding $w_{ij}v_j$ ranks the top among all the filters. Since v_j is the result of pooling function, which equals to the maximum value of the filter's original output vector O_j, we can trace back to find its position in vector O_j. Suppose this maximum value corresponds to the k_{th} value of vector O_j, then we can trace back again to locate the k_{th} n-gram sequence in the sentence. This n-gram sequence should be an important pattern to explain why this sentence is identified as feature request, since the corresponding mathematical transformation result of this sequence contributes the most to the final classification result.

As an example, we input the sentence "*Therefore, it would be nice if AdaDelta could be added to the set of available optimizers.*" into the trained CNN model, and analyze the intermediate outputs to locate the X-gram sequences of the top-10 filters. Among these filters, two of them have the height of 5, and their corresponding 5-gram sequences are *"it would be nice if"* and *"if AdaDelta could be added"*. Other filters are with smaller heights and most of their corresponding X-gram sequences are overlapping with the two 5-gram sequences mentioned above. From this example, we can see that CNN is also able to identify reasonable patterns in the form of key phrases in a sentence to interpret its classification result. As a future work, we plan to conduct a more formal study to improve interpretability of CNN's classification results.

9.2 Implications

Although our approach achieves a high accuracy when classifying sentences from developer discussions in issue tracking systems, the ultimate goal of intention mining is to leverage certain types of intentions for further analysis to solve software engineering tasks. In general, given a specific task, not all sentences in developer discussions are worthy for textual analysis, and intention mining can serve as a data pre-processing tool to filter out the noisy data.

As an example, we present a "Where should I post?" problem, that can benefit from intention mining. During our manual sentence labeling process, we found that a number of issue reports are directly closed by the project maintainer due to its unsuitable topic. For example, in TensorFlow Issue #773, the issue reporter asked *"Is there any way to get the gradients of activations (not the parameters), and watch them in Tensorboard?"*. From this question, we can see that he is seeking for help about TensorFlow's functionality. Thus, the maintainer closed this issue report and suggested to post his question on Stack Overflow, since it is not a bug, or a feature request. Both the issue reporter and the maintainer would waste their time due to the question asked in the wrong place. To address this problem, we can possibly design an automatic tool to analyze the textual description and give appropriate suggestions when a developer is asking general questions instead of reporting bugs or proposing feature requests. To do so, a possible solution is to apply our pre-trained CNN model to check whether the issue report contains *problem discovery* or *feature request* sentences. Additionally, we can extract *information seeking* sentences to perform further analysis, such as identifying whether the post is a general question or bug-related question. In future work, we would like to validate whether intention mining can be effectively applied to solve this problem.

9.3 Threats to Validity

Threats to internal validity relates to the errors in our code, the used taxonomy and the personal bias in manual classification of sentences. To reduce errors in our code, we have double checked and fully tested our code, still there could be errors that we did not notice. To reduce the impact of undetected errors in our code, we also published our source code and dataset to enable other researchers replicate and extend our work. To reduce subjectivity in the design of sentence taxonomy, we extended the taxonomy proposed by Di Sorbo et al.', which has been shown to be effective in analyzing developer discussions from mailing lists and user feedbacks from app reviews. To reduce the personal bias in manual annotation process, we strictly followed the card sorting process and had the first two authors involved label the sentences independently. The high Kappa level reported indicates a substantial agreement between the labelers. We further reduced bias by using an external evaluator to help resolve disagreements. These steps increase our confidence in the manually created dataset. We also provided the context of the sentence, such as the paragraph the sentence is in and the corresponding issue report, to avoid the difficulty in classifying sentences containing the word like "this", "that", "it", and "they". The user study on which we report later in this paper (see Section 7) also helps reduce threats by having professional developers assess whether or not the categories we produced using model learned from training data from this dataset are meaningful and appropriate. Another threat is that participants' degree of carefulness and effort in our user study may also affect the validity of our user study results. To reduce this threat, we recruited participants who expressed interests in our research and double checked the user study results to make sure there is no error due to inconsistency (i.e., the participant mistakenly labelled the entire category).

Threats to external validity relates to the generalizability of our results. Although we have collected a large number of sentences (i.e., 5,408), these sentences were sampled from a limited number (i.e., 100 or 200) of issue reports instead of all issue reports from each project, which might reduce the diversity of our dataset. Also, the number of sampled issues didn't follow a stratified sampling strategy (i.e., TensorFlow has the lowest number of total issues while we sampled 200 issues from TensorFlow), which might introduce bias to our experiment results. Besides, considering the huge number of projects hosted on GitHub, the experiment results achieved with the data extracted from only four projects may not be generalizable to developer discussions recorded in other projects. Moreover, although GitHub issue tracker is widely used, there are also other popular issue tracking systems (e.g., Bugzilla and JIRA), and it is unknown whether our approach still works well for developer discussions from these issue tracking systems. In a future work, we plan to collect more data from more projects hosted on different issue tracking systems for evaluation to mitigate this threat. On the other hand, we used cross-project prediction to evaluate our approach, and the results show that our approach can be trained from existing project and performs well when applied to a new project. However, when using data from issue reports to train our approach and predict data in emails, the prediction result is less accurate, which indicates that our approach does not fully bridge the gap between different communication channels. We plan to reduce this threat further by improving our approach further and analyzing more sentences from other forms of developer discussion.

Threats to construct validity relates to the suitability of our evaluation measures. We used accuracy, precision, recall and F1-score which are also used by Di Sorbo et al.'s study [10], and past studies to evaluate the performance of various automated software engineering techniques [73]–[77]. Thus, we believe there is little threat to construct validity.

10 Conclusion and Future Work

In this paper, we manually categorize 5,408 sentences from issue reports of four projects in GitHub, and refine Di Sorbo et al.'s taxonomy of intentions. We propose a deep learning based approach to automatically and more accurately classify sentences into different categories of intentions. Our approach integrates CNN with batch normalization to boost the training speed by at least 10 times. We also propose an automatic hyperparameter tuning approach, along with an ensemble learning component to further improve the accuracy of CNN with a reasonable training time cost. Our approach achieves an average accuracy of 0.8 on both Di Sorbo et al.'s and our dataset, which improves Di Sorbo et al.'s approach and the other automated sentence classification approaches by a substantial margin. To further investigate the generalizability of our approach, we conducted a user study with 11 professional developers and we find that 89% of the classification results are agreed by the majority of developers. Finally, we conducted a case study to show how our approach can serve as a support tool to help in rectifying misclassified issue reports.

In the future, we plan to refine our approach further by considering the context of a sentence, and leveraging multi-label learning algorithms [90] to classify a sentence into multiple intention categories simultaneously. We also plan to evaluate the performance of our approach with developer discussions from other communication channels, e.g., a code review system. Finally, we plan to apply our approach to downstream tasks. For example, since the F1-score achieved by our approach for *information seeking* is rather high (i.e., more than 0.9), we plan to leverage the identified sentences of information seeking to summarize frequently asked questions from the issue tracking system of a project. Then, we plan to summarize different solutions for these FAQs by extracting the sentences of solution proposal.

Acknowledgment. We would like to thank Di Sorbo et al. for sharing their replication package. We also thank the reviewers for their valuable comments. To enable other researchers to replicate and extend our study, we have published our source code and our dataset at https://github.com/tkdsheep/Intention-Mining-TSE.

References

[1] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol, "How developers' collaborations identified from different sources tell us about code changes," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 251–260.

[2] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 323–332.

[3] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 124–133.

[4] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 44.

[5] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto, "How the evolution of emerging collaborations relates to code changes: an empirical study," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 177–188.

[6] A. Begel and N. Nagappan, "Global software development: Who does it?" in *Global Software Engineering, 2008. ICGSE 2008. IEEE International Conference on*. IEEE, 2008, pp. 195–199.

[7] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 63–72.

[8] E. Knauss, D. Damian, J. Cleland-Huang, and R. Helms, "Patterns of continuous requirements clarification," *Requirements Engineering*, vol. 20, no. 4, pp. 383–403, 2015.

[9] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[10] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall, "Development emails content analyzer: Intention mining in developer discussions (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 12–23.

[11] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *Software maintenance and evolution (ICSME), 2015 IEEE international conference on*. IEEE, 2015, pp. 281–290.

[12] "Tensorflow," https://github.com/tensorflow/tensorflow.

[13] "Docker," https://github.com/moby/moby.

[14] "Bootstrap," https://github.com/twbs/bootstrap.

[15] "Vscode," https://github.com/Microsoft/vscode.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[17] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.

[18] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," *arXiv preprint arXiv:1702.05701*, 2017.

[19] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 321–332.

[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 448–456.

[21] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1746–1751.

[22] X. Gu and S. Kim, "What parts of your apps are loved by users?(t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015, pp. 760–770.

[23] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[24] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 49–60.

[25] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 602–611.

[26] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[27] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[28] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 531–540.

[29] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[30] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 499–510.

[31] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *ICSE 2017, to appear*.

[32] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 14–24.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[34] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 461–470.

[35] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 72–81.

[36] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 300–310.

[37] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 1074–1083.

[38] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 396–407.

[39] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 869–879.

[40] D. Hou and L. Mo, "Content categorization of api discussions," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 60–69.

[41] P. K. Prasetyo, D. Lo, P. Achananuparp, Y. Tian, and E.-P. Lim, "Automatic classification of software related microblogs," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 596–599.

[42] F. Thung, D. Lo, M. H. Osman, and M. R. Chaudron, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 110–121.

[43] 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 *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 476–481.

[44] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 404–415.

[45] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[46] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[47] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 51–62.

[48] G. Chen, C. Chen, Z. Xing, and B. Xu, "Learning a dual-language vector space for domain-specific cross-lingual question retrieval," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 744–755.

[49] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[50] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[51] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[53] I. Lenz, H. Lee, and A. Saxena, "Deep learning for detecting robotic grasps," *The International Journal of Robotics Research*, vol. 34, no. 4-5, pp. 705–724, 2015.

[54] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[55] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.

[56] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[57] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[58] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[59] T. Hofmann, "Probabilistic latent semantic indexing," in *ACM SIGIR Forum*, vol. 51, no. 2. ACM, 2017, pp. 211–218.

[60] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[61] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 92–96.

[62] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.

[63] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[64] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[65] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[66] R. Hecht-Nielsen *et al.*, "Theory of the backpropagation neural network." *Neural Networks*, vol. 1, no. Supplement-1, pp. 445–448, 1988.

[67] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," *arXiv preprint arXiv:1510.03820*, 2015.

[68] "Implementation of kim's cnn based on tensorflow," https://github.com/dennybritz/cnn-text-classification-tf.

[69] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[70] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep image: Scaling up image recognition," *arXiv preprint arXiv:1501.02876*, vol. 7, no. 8, 2015.

[71] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[72] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[73] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 432–441.

[74] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 279–289.

[75] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 382–391.

[76] L. Guo, B. Cukic, and H. Singh, "Predicting fault prone modules by the dempster-shafer belief networks," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 2003, pp. 249–252.

[77] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[78] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[79] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 812–823.

[80] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[81] J. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods - Support Vector Learning*, B. Schoelkopf, C. Burges, and A. Smola, Eds. MIT Press, 1998. [Online]. Available: http://research.microsoft.com/~jplatt/smo.html

[82] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[83] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 826–830, 2017.

[84] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," 2012.

[85] S. Bird, "Nltk: the natural language toolkit," in *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics, 2006, pp. 69–72.

[86] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[87] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[88] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[89] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[90] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 8, pp. 1819–1837, 2014.