11-2021

Automating developer chat mining

Shengyi PAN
Zhejiang University

Lingfeng BAO
Zhejiang University

Xiaoxue REN
Zhejiang University

Xin XIA
Monash University

David LO
Singapore Management University, davidlo@smu.edu.sg

See next page for additional authors

Citation

PAN, Shengyi; BAO, Lingfeng; REN, Xiaoxue; XIA, Xin; LO, David; and LI, Shanping. Automating developer chat mining. (2021). *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE): Melbourne' November 14-20: Proceedings*. 845-866.
Available at: https://ink.library.smu.edu.sg/sis_research/6809

Automating Developer Chat Mining

Shengyi Pan*†, Lingfeng Bao*¶, Xiaoxue Ren*, Xin Xia‡, David Lo§, Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
‡Faculty of Information Technology, Monash University, Australia
§School of Information Systems, Singapore Management University, Singapore
shengyipan@outlook.com, {lingfengbao,xxren,shan}@zju.edu.cn, Xin.Xia@monash.edu, davidlo@smu.edu.sg

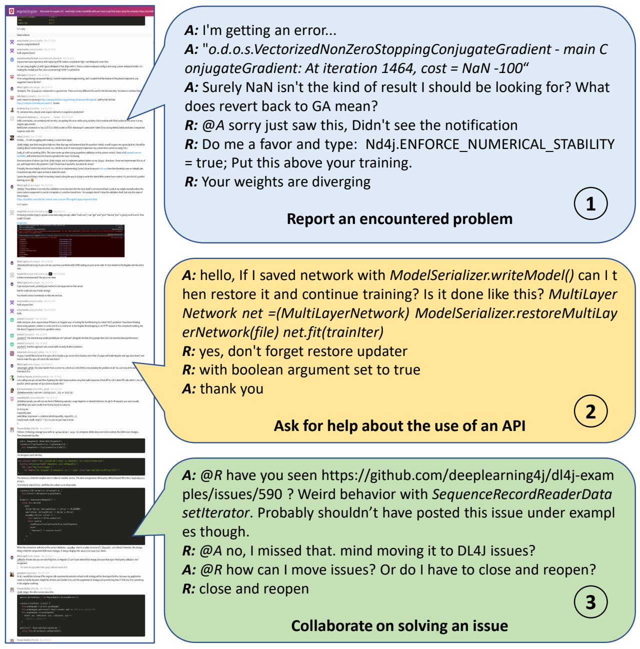
***Abstract*—Online chatrooms are gaining popularity as a communication channel between widely distributed developers of Open Source Software (OSS) projects. Most discussion *threads* in chatrooms follow a Q&A format, with some developers (*askers*) raising an initial question and others (*respondents*) joining in to provide answers. These discussion threads are embedded with rich information that can satisfy the diverse needs of various OSS stakeholders. However, retrieving information from threads is challenging as it requires a thread-level analysis to understand the context. Moreover, the chat data is transient and unstructured, consisting of entangled informal conversations. In this paper, we address this challenge by identifying the information types available in developer chats and further introducing an automated mining technique. Through manual examination of chat data from three chatrooms on Gitter, using card sorting, we build a thread-level taxonomy with nine information categories and create a labeled dataset with 2,959 threads. We propose a classification approach (named F2CHAT) to structure the vast amount of threads based on the information type automatically, helping stakeholders quickly acquire their desired information. F2CHAT effectively combines handcrafted non-textual features with deep textual features extracted by neural models. Specifically, it has two stages with the first one leveraging the siamese architecture to pretrain the textual feature encoder, and the second one facilitating an in-depth fusion of two types of features. Evaluation results suggest that our approach achieves an average F1-score of 0.628, which improves the baseline by 57%. Experiments also verify the effectiveness of our identified non-textual features under both intra-project and cross-project validations.**

***Index Terms*—Developer Chatrooms, Information Mining, Deep Learning, Gitter**

I. INTRODUCTION

Recent studies showed that online chatrooms are gaining popularity as a channel for global collaboration among developers of Open Source Software (OSS) projects and replacing the traditional communication platforms, including emails and mailing lists [1]–[3]. Chatrooms like Gitter [4], Slack [5], and Discord [6], are modern instant messaging systems integrated with diverse external services (e.g., bots, issue linking), making developers easier to communicate and collaborate with others [7]. Developers use chatrooms to report problems, share opinions, and discuss implementation details [8], [9].

Most discussion *threads* in developer chatrooms generally follow a Q&A format [8], [9], with some developers (*askers*) raising an initial question and others (*respondents*) later joining in to provide answers. Previous studies [7]–[11] have revealed


A: I'm getting an error...
A: "*o.d.o.s.VectorizedNonZeroStoppingConjugateGradient - main C onjugateGradient: At iteration 1464, cost = NaN -100*"
A: Surely NaN isn't the kind of result I should be looking for? What does revert back to GA mean?
R: @A sorry just saw this, Didn't see the notification
R: Do me a favor and type: Nd4j.ENFORCE_NUMERICAL_STABILITY = true; Put this above your training.
R: Your weights are diverging

A: hello, If I saved network with *ModelSerializer.writeModel()* can I t hen restore it and continue training? Is it done like this? *MultiLayer Network net =(MultiLayerNetwork) ModelSerializer.restoreMultiLay erNetwork(file) net.fit(trainIter)*
R: yes, don't forget restore updater
R: with boolean argument set to true
A: thank you

A: @R have you seen https://github.com/deeplearning4j/dl4j-exam ples/issues/590 ? Weird behavior with *SequenceRecordReaderData setIterator*. Probably shouldn't have posted this issue under exampl es though.
R: @A no, I missed that. mind moving it to DL4J issues?
A: @R how can I move issues? Or do I have to close and reopen?
R: close and reopen

Fig. 1: Example discussion threads with three different intents from Deeplearning4j chatroom on Gitter. ***A*** and ***R*** represent the asker and the respondent, respectively.

that these discussion threads are embedded with rich information, which can be utilized to support various development activities. Conversations in developer chatrooms are typically informal, with rapid exchanges of messages between two or more participants and fewer constraints on discussion topics. Compared with other communication channels, e.g., mailing lists and issue tracking systems (ITS), both types and amount of valuable information in chatrooms are much richer [9], [12].

The needs to acquire information from discussions in chatrooms vary among different OSS stakeholders based on their tasks and interests. For development teams, they may want to be aware of the problems reported by end-users to deliver an early fix to a bug. They may also want to monitor the progress of tasks towards the resolutions of issues to facilitate better collaboration. For end-users, they may care more about solutions and opinions in previous discussions related to their problems. Similar questions are usually repeated many times by the community as the chat conversations are generally short-lived [9], quickly flooded away by the incoming messages. Figure 1 illustrates that information embedded in discussion threads is

†Also with PengCheng Laboratory.
¶Corresponding author.

capable of fulfilling the needs of various OSS stakeholders. Hence, a comprehensive analysis of information categories available in developer chats is of vital importance.

Retrieving information from massive chat data requires a huge effort, as some discussions could be very lengthy and hard to follow. Automated mining techniques are urgently needed to collect various information embedded in massive chat data and categorize it properly, helping OSS stakeholders directly retrieve their desired information from the well structured data. However, there are several hurdles that may prevent an effective mining. 1) **Thread-level analysis.** Different from prior mining tasks [11], [13]–[15] focusing on sentence-level classification, thread is a natural granularity for mining information from developer chatrooms as the smallest unit containing comprehensive context information. A sentence-level analysis abandoning the context prevents an accurate interpretation during classification, and faces lots of short and incomplete sentences (see examples in Figure 1). 2) **Noisy data.** Chatrooms have multiple participants who take part in different discussions at the same time, thus disentangling threads from the stream of messages is critical for enabling a thread-level analysis. Additionally, informal conversations contain many short and meaningless instant messages, as well as typos and special tokens (e.g., code snippet, URL), which greatly affect the performance of text classification techniques. To the best of our knowledge, the approach proposed in [12] (named FRMiner) is the only one targeted towards mining information from developer chats at thread level. However, FRMiner is only focused on detecting one specific type of discussion threads, i.e., threads with hidden feature requests, to support the release team, while ignoring other valuable information and different interests of other OSS stakeholders.

The machine learning (ML) based mining techniques in previous studies typically rely on shallow textual features (e.g., TF-IDF, bag-of-words) [16], [17], non-textual features (e.g., sentence length, time gap) [18], [19] or the combination of both features [13], [20]. Specially, Arya *et al.* [13] and Wood *et al.* [20] reported that non-textual features are more useful compared with the textual counterpart under certain circumstances. The deep learning (DL) based approaches in recent studies [12], [21] leverage advanced neural models to extract deep textual features, which are powerful representations with high-level semantic information. However, we argue that the textual features ignore other information of the discussion thread (e.g., structure, participant). Thus, the DL-based approaches may still benefit from the handcrafted non-textual features in this specific task.

In this work, we take the first step to analyze the information types available in discussion threads from developer chatrooms. We aim to reveal the characteristics of information types, their primary intents, and possible applications, which are essential for designing automated mining techniques to fulfill the needs of various OSS stakeholders. Through manual examination of historical chat data (2,959 threads) from three chatrooms on Gitter, using card sorting, we build a thread-level taxonomy with nine information categories. Further, we propose a classification approach, namely F2CHAT, which effectively combines handcrafted non-textual **F**eatures with deep textual **F**eatures extracted by neural models. Specifically, it has two stages with the first one leveraging the siamese architecture to pretrain the textual feature encoder, and the second one facilitating an in-depth fusion of two types of features. We evaluate our approach on 2,959 threads labeled using the defined taxonomy. The experimental results indicate that our approach improves the performance of FRMiner by 57%, with an overall F1-score of 0.628. The experiments also verify the effectiveness of our identified non-textual features under both intra-project and cross-project validations.

The contributions of our work are summarized as follows:

- We are the first to build a thread-level taxonomy of information types for threads in developer chatrooms.
- We propose an automated mining approach (F2CHAT), which combines handcrafted non-textual features with deep textual features extracted by neural models.
- We conduct extensive experiments to evaluate F2CHAT on three chatrooms from Gitter. The experimental results indicate that our approach substantially outperforms FR-Miner and the identified non-textual features are effective in both intra-project and cross-project validations.
- We open source our replication package and a dataset of 2,959 discussion threads [22], annotated with the identified taxonomy of information categories.

II. RELATED WORK

A. *Developer Online Chatrooms*

Previous studies reported that online chatroom plays an increasingly important role in various development activities, and tried to understand the way of developers using chatrooms by analyzing their behaviors and interactions. Shihab et al. [23], [24] reported that there is a shift from mailing lists to developer Internet Relay Chat (IRC) meetings, and further investigated the role of IRC meetings using two open source projects. Lin *et al.* [1] conducted an exploratory study to investigate the way of developers using Slack. Their findings suggested that Slack is used for personal, team-wide, and community-wide purposes, and is gradually replacing the emails. Sahar *et al.* [7] assessed the impact of Gitter on project and team dynamics. They focused on issue report discussions and found that they are closely related to activities in the GitHub issue tracker. Ehasan *et al.* [8] conducted a study to understand the general Q&A behaviors of discussions in Gitter. They are the first to propose an automatic approach for thread identification in developer chatrooms and further explore the nature of discussions (e.g., topics) by performing a thread-level analysis. These two works suggested that Gitter chatrooms are rich sources for information related to the software.

Other works focused on mining information from developer chats to support software development and maintenance. Alkadhi *et al.* [10], [11] conducted exploratory studies to examine the frequency and completeness of rationale hidden in chat messages. They found that chat messages are a valuable source for rationale and the machine-learning based algorithms are capable of automatically extracting rationale from IRC messages. Chatterjee *et al.* [9] pointed out that conversations in developer chatrooms generally follow a Q&A format and investigated the types, amount and possible mining hurdles of information in Slack Q&A chats compared with Stack Overflow Q&A posts. They also emphasized the importance of a thread-level analysis, and later released a dataset of software related conversations from Slack with a customized disentanglement algorithm in [25]. Recently, Shi *et al.* [12] designed a deep-learning model to detect threads with hidden feature requests. Their experimental results on three OSS chatrooms from Slack suggested that their method outperforms the existing sentence-level methods by a large margin.

Most prior works are exploratory studies, aiming to understand the role of chatrooms in supporting developers or investigate the validity of chatrooms as a mining source. Our work differs from these studies. We take a step forward to perform an in-depth thread-level analysis for identifying information types available in developer chats on Gitter. Additionally, we design a classification approach and evaluate its effectiveness in detecting information types of discussion threads.

B. *Classification of Software-related Artifacts*

In recent years, an increasing number of studies have focused on mining information from software-related artifacts by classifying them into categories relevant to various software activities. The automated mining techniques proposed by prior works can be generally categorized into three groups according to the leveraged features: 1) **Textual features.** Bacchelli *et al.* [16] represented lines of development emails as vectors of term frequencies (TF) and applied machine-learning (ML) techniques to classify them into five categories (nature language, source code, patch, stack trace, and junk) based on the specific content. However, Di Sorbo *et al.* [14] and Panichella *et al.* [15] argued that techniques based on lexicon analysis, such as Vector Space Models [26] (e.g., TF-IDF, bag-of-words) and topic models (e.g., LDA), would not be sufficient as they failed to reveal the developers' intents. To bridge the gap, they applied heuristics to capture linguistic patterns for the classification of sentences in emails [14] and app reviews [15] based on developers' purposes. Recently, deep learning (DL) approaches have been introduced in this task, which automatically learn linguistic patterns and are more powerful in extracting high-level semantic information. Huang *et al.* [21] applied textCNN [27] to classify sentences in both emails and ITSs. Their method substantially outperformed Di Sorbo *et al.*'s heuristics [14] and five ML based text classification techniques using Vector Space Model. Shi *et al.* [12] proposed a novel approach to enable a thread-level analysis for detecting hidden feature requests from discussions on Slack. Besides, They are the first to incorporate few-shot learning techniques [28] to make the maximum use of limited labeled data. 2) **Non-textual features.** Rastkar *et al.* [18] identified 24 non-textual features from four aspects (structure, participant, length and lexicon), and trained logistic regression (LR) classifiers for automatic summarization of bug reports. Similar non-textual features were also utilized in [19] to generate summaries for developer-client conversations. 3) **Both textual and non-textual features.** Wood *et al.* [20] utilized bag-of-words as the textual feature together with three non-textual features to train LR classifiers for detection of 26 speech act types in conversations during bug repair. They reported that non-texture features are very useful for certain speech act types. Arya *et al.* [13] leveraged several ML algorithms for classifying sentences in issues. They applied TF-IDF weights as the textual feature and identified 14 non-textual features. They found that using non-textual features alone is even better than both features under intra-project validation. The shallow textual features used in these two works only contain lexical information. Moreover, the fusion of features is simple as inputting to the ML classifiers simultaneously.

Our work differs from the existing studies. To the best of our knowledge, we are the first to combine handcrafted non-textual features with deep textual features extracted by neural models for classification of software-related artifacts. We also propose an architecture to facilitate an in-depth fusion of two types of features. Furthermore, while most of the prior works performed classification at the granularity of sentences, we focus on a thread-level analysis of developer discussions.

III. Analyzing Information Categories of Discussion Threads

In this section, we try to identify potential information types available in developer chats that may satisfy the diverse needs of various OSS stakeholders. We first introduce the four steps used to collect chat data from Gitter chatrooms. Then, we describe the details of identifying information types.

A. *Data Preparation*

Step 1: Chatroom Selection. we select three chatrooms on Gitter: Angular [29], Spring-boot [30] and Deeplearning4j [31]. We choose these chatrooms for the following reasons: 1) These chatrooms belong to three different categories. Gitter divides all its chatrooms into 24 categories [32], e.g., Frontend, Android, and Data science. Choosing chatrooms of different categories strengthens the generalizability of our work by allowing us to investigate chat data of diverse development topics. 2) Large numbers of developers actively communicate with each other in these chatrooms. We sort all the chatrooms on Gitter based on the number of participants since it is one of the attributes supported by Gitter API. We further exclude chatrooms with limited chat data (less than 50,000 messages) by crawling and counting all of its historical messages. Finally, we select the top three chatrooms under the premise of belonging to different categories. The characteristics of these chatrooms are presented in Table I.
Step 2: Data Crawling. We crawl the historical chat data of the selected chatrooms using the official API [33] provided by Gitter. We crawled data on August 20, 2020.
Step 3: Thread Disentanglement. In chatrooms, multiple developers participate in different threads at the same time.

TABLE I: The characteristics of the selected chatrooms

Project	Categories	#Participants	#Messages	Time Duration
Angular	Javascript	22,771	1,120,438	2015.03-2020.08
Spring-boot	Java	9,665	70,590	2014.10-2020.08
Deeplearning4j	Data Science	8,356	415,422	2015.03-2020.08

Two latest disentanglement algorithms targeted for technical discussions in developer chatrooms are [8], [25]. Chatterjee *et al.* [25] modified Elsner and Charniak's algorithm [34] for the customization of several Slack specific features. Ehsan *et al.* [8] identified three categories of features (i.e., users, content and back-and-forth communication) through manual analysis of Gitter chats and futher designed a heuristic-based algorithm. We apply Ehasan *et al.*'s algorithm to disentangle the distinct threads from the stream of messages, since it was demonstrated to achieve satisfactory results on Gitter chats with an F1-score of 0.81 [8]. While Chatterjee *et al.*'s algorithm still requires to be adapted to work well on chat platforms other than Slack, including Gitter used in our study, as suggested in [25].

Step 4: Thread Sampling. We randomly sample 1,000 threads from each chatroom and manually exclude low-quality threads of the following characteristics: 1) Threads that contain too much unformatted source code or stack traces. 2) Threads with too many spelling and grammatical errors or written in non-English languages. The number of available threads for each chatroom is shown in column *THD* of Table IV.

B. Building Taxonomy of Information Types

The sentence-level information types have been widely studied in prior works regarding the classification of development communication artifacts [13], [20], [21]. But they are not applicable to threads with multiple messages, as they focus on the content of an individual sentence while abandoning the context. However, we manage to utilize the knowledge in two latest sentence-level works: 1) Arya *et al.* [13] identified 16 information types (e.g., Bug Reproduction and Solution Discussion) for discussions in ITSs based on the underlying purposes of users. We build our thread-level taxonomy by summarizing their sentence-level information types. 2) Wood *et al.* [20] uncovered 26 speech act types (e.g., Documentation Answer and API Question) in conversations during bug repair based on the specific discussed development contents. Their work inspires us to build a taxonomy from the aspect of discussion contents. Di Sorbo *et al.* [14] and Panichella *et al.* [15] reported that taxonomies in related works are generally defined from two aspects: text contents or developers' purposes.

Besides, we follow the card sorting process [35] to identify the information types of discussion threads, which has been proven effective in previous works, e.g., identifying intentions of sentences in ITS [21]. We created one card for each sampled thread. The first two authors worked together to determine the label of each card. The whole process has two iterations:

Iteration 1. We first used 20% of the cards. The two authors coded each thread individually to identify possible categories at thread level. They first read all messages in the thread and identified information type for each one using the taxonomy described in [13]. Then, they summarized potential commonalities of the sentence-level information types and inferred the underlying purpose of the discussion. After that, they focused on the specific development contents discussed in the thread. Finally, they worked together to discuss the findings and disagreements. In this iteration, we built a two-level taxonomy (Table II) along with a handbook to guide classification based on the following findings:

Level 1: We summarize three intents of developer discussions by leveraging the 16 sentence-level information types in [13].

- **Problem Report.** Askers (mostly the project end-users) report bugs or describe unexpected behaviors by providing source code, full stack traces, or specifying what they have tried to do. Respondents try to help askers find the cause of their problems, guide them in the right direction, or share possible solutions. See ① in Figure 1 for an example.
- **Information Retrieval.** Askers (mostly the project end-users) attempt to obtain information or help from the community about API usage, library installation, documentation resources, etc. Usually, askers describe what they want to accomplish and try to get some help before they dive into the specific development. Respondents share opinions about the best practice or provide suggestions of implementation details. See ② in Figure 1 for an example.
- **Project Management.** Although most discussion threads in developer chatrooms are initiated with end-users asking questions about the use of the software, team members and certain users (potential contributors) also collaborate through online chatting to solve issues filed on GitHub. For example, working together to identify causes and solutions, discussing the testing procedure and results, and requesting or reporting the progress of tasks. Besides, the team also announce the release of a new version and answer questions regarding the future roadmap of the software evolution in online chatrooms. See ③ in Figure 1 for an example.

Level 2: The specific development contents discussed in online chatrooms can be categorized into nine classes as shown in column *level 2* of Table II. We find the development contents discussed in chatrooms are generally consistent with those identified by [20] in chat conversations during bug repair. However, we add a sub-class *General Information* under *information Retrieval*, since there are some social discussions that are not closely related to the project itself, e.g., the best choice of IDE, job hunting experience, and the design of deep learning models. Besides, we divide *Project Management* into two sub-classes: 1) *Task Progress.* The development team and end-users request or report task progresses, inquire or inform release plans, and discuss future roadmap. 2) *Technical Discussion.* Discussions of this class focus more on technical parts, where participants collaborate to find the cause of issues, analyze the testing results and seek possible solutions.

Iteration 2. Two authors independently labeled the remaining cards into nine categories listed in Table II. We use Cohen's Kappa coefficient [36] to measure the agreement between two authors. Their Kappa value for nine categories

TABLE II: The built taxonomy of information categories for developer discussion threads

Level 1	Description	Level 2	Description
Problem Report	**Asker:** report bugs or describe unexpected behaviors. **Respondent:** help the asker find the cause of the problem and provide possible solutions.	Programming Problem (PP)	problems related to programming, e.g., syntax, parameter, API and implementation
		Library Problem (LP)	problems related to library installation, deployment and configuration
		Documentation Problem (DP)	problems related to documentation resources, e.g., examples, guidance and docs
Information Retrieval	**Asker:** try to obtain information or help from the community regarding some development issues. **Respondent:** share opinions about the best practice or suggestions of implementation details.	Programming Information (PI)	information related to programming, e.g., syntax, parameter, API and implementation
		Library Information (LI)	information related to library installation, deployment and configuration
		Documentation Information (DI)	information related to documentation resources, e.g., examples, guidance and docs
		General Information (GI)	information related to general knowledge, e.g., job hunting experience and choices of IDEs
Project Management	**End-user:** request the progress of certain issues or the release schedule of the project. **Team:** collaborate on solving issues and inform community about future plans.	Technical Discussion (TD)	technical discussions during collaboration on solving issues, e.g., finding causes and solutions
		Task Progress (TP)	communications on task progresses, release schedules and future plans

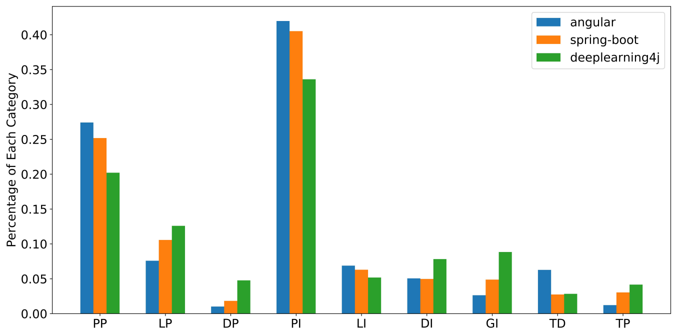


Fig. 2: Distribution of the percentage of each information type in the sampled discussion threads

of level 2 is 0.76, which is lower than the one (0.83) for three categories of level 1. Both Kappa values indicate a substantial agreement between the two authors. For cards with disagreements, two authors discussed to reach a common decision and further refined the guidance handbook for classification.

After two iterations of card sorting, we labeled 2,959 discussion threads from three chatrooms on Gitter. The annotation is extremely expensive, with a cost of 480 person-hours. The distribution of each information type is shown in Figure 2. Over 80% of the discussion threads in all three chatrooms belong to the category *Problem Report* or *Information Retrieval*, and majority of them are related to programming (e.g., API, syntax and parameter). It indicates that developers mainly use online chatrooms to seek solutions or answers for their problems.

IV. AUTOMATED CLASSIFICATION OF INFORMATION CATEGORIES

In this section, we first describe the detailed process for preprocessing the chat messages. Then, we build a two-stage model (Figure 3), namely F2CHAT, for automated classification of information categories.

A. Preprocessing of Chat Messages

Messages in developer chatrooms are noisy for text classification algorithms (as described in Section I). A throughout data preprocessing is critical for automated techniques to achieve satisfactory performances. In this work, we take the following steps to preprocess the raw chat messages:

1) **Fine-grained special tokens replacement.** The development chat messages contain lots of special tokens such as source code, URL, and issue ID. To clean the sentences, we replace these tokens with specific tags (e.g., *CODETAG*, *URLTAG* and *ISSUETAG*) using regular expressions. This step is widely adopted in the related works [12], [13]. However, we argue that the replacements in previous studies are not precise enough. Specifically, special tokens should be replaced based on the specific artifacts contained rather than their forms. For example, a URL can link to an issue on Github, a page of the official docs, etc. These differences should not be ignored, i.e., we should not replace all URLs with a single tag. Moreover, the same artifact is referred to using different forms (e.g., an issue can be referred to using its ID or a direct URL link). This diversity (e.g., replacing issues with different tags) can confuse the model. We perform a fine-grained special tokens replacement leveraging the knowledge from manual examination of chat data.
2) **Merge consecutive messages from the same developer.** Unlike the well-structured discussion threads in issue tracking systems, developer chats are generally informal conversations with lots of quick and incomplete messages. While some developers prefer to comprehensively describe his problems and thoughts within a single long message (②, ③ in Figure 1), others may prefer a series of short messages instead (① in Figure 1). This inconsistency caused by the personal habits of different developers will bring

troubles when we calculate certain non-textual features (e.g., number of messages within a thread) or model the message sequence. We merge consecutive messages from the same developer to eliminate this inconsistency.

B. Pretraining of Textual Feature Encoder

We separate the training of encoders for two types of features (Figure 3) due to the following concerns: 1) Textual features and non-textual features have different data types, i.e., high-dimensional sparse vectors vs. numerical values (Table III), training two encoders (one for each type of features) simultaneously (let alone a unified encoder) is not the best practice. 2) Textual features are encoded by deep neural models with large amounts of parameters, thus few-shot learning techniques are needed to overcome the overfitting problem. It is not the case for non-textual features since they are encoded using a two-layer feed-forward module. 3) Recasting a classification task into similarity measurement will cause information loss [37].

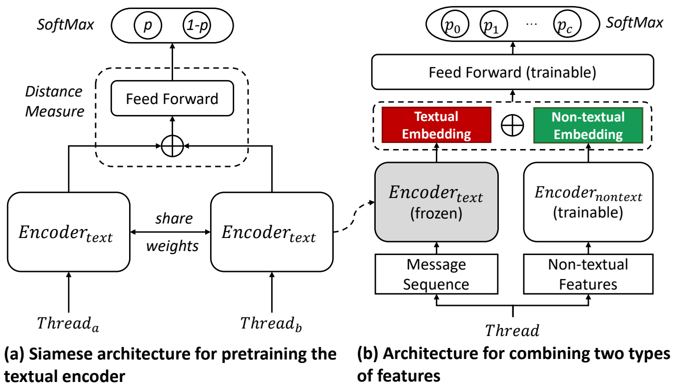


Fig. 3: Two-stage model architecture of F2CHAT

The first stage of our proposed two-stage classification approach focuses on pretraining the textual feature encoder. We refer to the model at this stage as F2CHAT-t, since it only leverages textual features. Given the fact that the annotation is extremely expensive (Section III-B), we follow the methods proposed by Shi *et al.* [12] to alleviate the overfitting problem caused by the insufficient data. By incorporating Siamese Network [38], a metric-based few-shot learning technique [28], they recast the traditional text classification task of classifying a single thread to the correct class into the task of determining whether a pair of threads belong to the same class or not. The siamese architecture for pretraining the textual encoder is shown in Figure 3(a). A pair of threads (either from the same class or different classes) are sent into two identical encoders (same structure and parameters) separately to get the textual feature embeddings $e = Encoder_{text}(t)$. The embeddings e_a and e_b are then used to measure the distance between two threads in the latent space. Specifically, we follow the metric in [12] to use a two-layer fully connected feed-forward network for distance measurement with the concatenation of two embeddings $e_{ab} = e_a \bigoplus e_b$ as the input. Finally, the distance embeddings are sent to a two-unit softmax layer to get the normalized score $[score_{same}, score_{diff}]$, indicating whether the two threads belong to the same class or not.

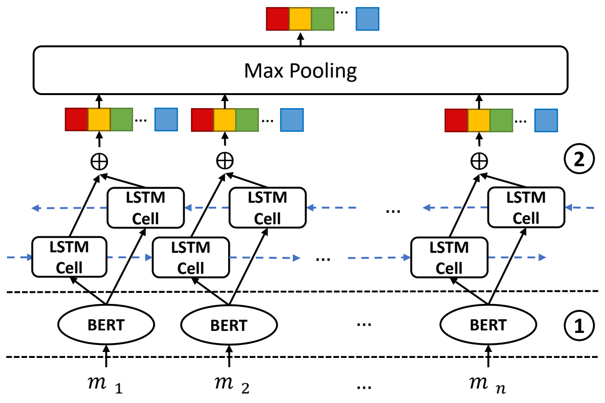


Fig. 4: The detailed structure of textual feature encoder

The detailed architecture of $Encoder_{text}$ is presented in Figure 4, which contains following two modules:

Message encoding module. We first encode each message in the thread (① in Figure 4). We utilize BERT [39], one of the state-of-the-art pretrained models (PTMs), to extract the textual representation for each message. Compared with the traditional context-free word embeddings (e.g., GloVe [40]), BERT aims to learn contextual word embeddings (i.e., the embedding of a word changes dynamically based on the context where it appears) from large unlabeled corpora, thus substantially boosting the performance of multiple natural language processing (NLP) tasks [41]. Besides, applying BERT in downstream tasks follows a simple fine-tuning process, which prevents us from diving into the design of sophisticated model architectures and introduces minimal task-specific parameters, which is important considering only a small dataset is available in our task. According to [39], we use the embedding of `[CLS]` (a special token in BERT which is added in front of every input sample) as the representation of the whole message for our classification task. BERT uses a subword tokenizer that avoids the out-of-vocabulary (OOV) problem by decomposing OOV words into known subwords. However, it does not apply to the technical terms in developer communications. Hence, we add several domain-specific terms (e.g., *maven*, *npm*, and *stackblitz*) to the original BERT vocabulary based on frequency. The embeddings for these terms will be optimized during fine-tuning.

Thread encoding module. We then model the contextual information of the entire thread from a sequence of message embeddings (② in Figure 4), as the semantic logic is a crucial pattern for thread-level classification. Bidirectional Long Short Term Memory Network (Bi-LSTM) [42] is utilized to capture the bidirectional contextual information for this sequence learning task, which stacks two standard LSTM layers with each one responsible for learning one-direction representation. We concatenate the outputs of Bi-LSTM $h = [\overrightarrow{h} \bigoplus \overleftarrow{h}]$ in each time step and further utilize a max-pooling layer to get the final embedding of the thread.

TABLE III: List of non-textual features

Type	Feature[1]	Description	Aspect[2]
Length	$token_a$	length of thread / first message / messages from the asker (number of tokens)	T,F,A
	$token_r$	length of first message / messages from the asker divided by thread length (number of tokens)	F,A
	$character_a$	length of thread / first message / messages from the asker (number of characters)	T,F,A
	$character_r$	length of first message / messages from the asker divided by thread length (number of characters)	F,A
Structural	$timegap_a$	thread duration / gap between the first and second message (time)	T,F
	$timegapMean_a$	mean of all gaps in the thread (time)	T
	$timegapStd_a$	standard deviation of all gaps in the thread (time)	T
	$timegap_r$	gap between the first and second message divided by thread duration (time)	F
	$messgap_a$	thread duration / gap between the first and second message (number of messages)	T,F
	$messgapMean_a$	mean of all gaps in the thread (number of messages)	T
	$messgapStd_a$	standard deviation of all gaps in the thread (number of messages)	T
	$messgap_r$	gap between the first and second message divided by thread duration (number of messages)	F
	$mess_a$	number of messages within the thread / from the asker	T,A
	$mess_r$	number of messages from the asker divided by total number of messages in the thread	A
Participant	$participant_a$	number of participants involved in the thread	T
Special-token	$questionmark_a$	number of question marks in the first message	F
	$greeting_a$	number of greeting words (e.g., hello, hey and hi) in the first message	F
	$code_a$	number of code snippts in thread / first message / messages from the asker	T,F,A
	$error_a$	number of stack traces in thread / first message / messages from the asker	T,F,A
	doc_a	number of documentations mentioned in thread / first message / messages from the asker	T,F,A
	$issue_a$	number of issues mentioned in thread / first message / messages from the asker	T,F,A

[1] Footnote a and r denote the features with absolute value ranged in $(0, +\infty)$ and the features with relative value ranged in $(0, 1]$, respectively.
[2] *T*, *F* and *A* denote the aspects of whole thread, first message and the asker, respectively.

C. Incorporating Non-textual Features

The second stage of F2CHAT focuses on the training of non-textual feature encoder and the combination of both textual and non-textual features. We first introduce the identified non-textual features and then describe the details of model architecture for stage two.

Handcrafted non-textual features. The non-textual features used in [13], [19], [20], [43] are generally based on the features first proposed by Murray and Carenini [44]. However, these features are targeted for classification at the granularity of sentence, thus they need to be adapted and further enriched for our thread-level algorithm. Inspired by the sentence-level features, we identify 21 features (Table III) which can be categorized into four groups:

1) **Length Features** refer to the features measured by absolute or relative (w.r.t. the entire thread) length in either token-level or character-level. Rationale: Threads of *Problem Report* are generally longer as the askers need to elaborate the details of the encountered problems, while expressing the needs of certain information in threads of *Information Retrieval* is relatively straightforward.
2) **Structural Features** refer to the features that are related to the structure of a thread, e.g., the time interval between first and last message of the thread (i.e., the duration of the discussion). We also measure through the number of intermediate messages since the time interval can be easily affected by external factors, e.g., the activeness of the chatroom which is different with respect to the specific time period [8]. Rationale: Threads of *Problem Report* usually have more messages and longer lasting time, since solving a problem usually requires an in-depth discussion to figure out the causes and takes time to verify the validity of potential solutions.
3) **Participant Feature** is the number of participants involved in the thread. Rationale: Although most of the discussions are between two participants, i.e., the asker and one respondent, more respondents will participate when the reported bugs are tricky (*Problem Report*) or a team collaboration is required (*Project Management*).
4) **Special-token Features** refer to the features measured by the number of special tokens (e.g., code snippets, stack traces and URLs). Rationale: Some special tokens are strong indicators of certain information categories, e.g., stack traces for *Programming Problem* and URLs (linking to official docs) for *Documentation Information*.

Furthermore, these 21 features are measured from three different aspects (column *Aspect* in Table III): 1) **Features related to the whole thread.** These features characterize the thread as a whole. 2) **Features related to the first message.** The first message is important as it usually indicates the purposes and requirements of the thread initiator (asker). For example, a large relative length of the first message w.r.t. the entire thread suggests that the thread is most likely to belong to *Problem Report*, since the asker needs to fully elaborate the encountered problems. 3) **Features related to the asker.** These features characterize the behaviour of the asker in the discussion thread. By extracting features from this aspect, we want to emphasize the differences between the two roles (asker and respondent) of the participants. For example, if a large portion of code snippets in the discussion thread are provided by the asker, then it is a strong indicator of *Problem Report*. However, this may not be the case if provided by the respondents, since respondents in the threads of *Information Retrieval* sometimes need to share the implementation details while the askers just simply express their needs.

Combining both textual and non-textual features. Figure 3(b) presents the model architecture for combining the two types of features in stage two. Unlike the textual counterpart, non-textual features are numerical values (Table III) that can be directly computed. There is no need for a complex encoder with a large number of parameters or few-shot learning techniques. We use a fully connected feed-forward network with one hidden layer to encode the non-textual features $e_{nontext} = Encoder_{nontext}(t)$, while the embedding of the textual features is extracted using the pretrained encoder from stage one (Figure 3) $e_{text} = Encoder_{text}(t)$. We set an extremely small learning rate (1e-5) for parameters in the pretrained textual feature encoder, since we want to avoid collapsing the well-trained encoder while slightly fine-tune it to accommodate the newly added non-textual features. Considering that some patterns require an in-depth integration of both textual and non-textual features, we perform an early fusion at the feature level instead of a late fusion at the decision level. We concatenate the embeddings of textual and non-textual features as the final representation of the thread $e = [e_{text} \bigoplus e_{nontext}]$ and then use it for decision making, instead of making two decisions using textual and non-textual features respectively and then fuse them to get the final decision. We use a feed-forward network for decision making. Finally, a softmax layer is utilized to get the normalized score of each information type for this multi-class classification task.

V. Experiment Design & Results

A. Experiment Settings

We use the dataset built in Section III for evaluation. The detailed statistics presented in Table IV include the number of sampled threads (THD), the median of the thread lasting time (DUR), the average number of participants per thread (PCP), the average number of messages per thread (MSG), and the average number of tokens per message (MSGLEN). The experimental environment is a server equipped with an NVIDIA V100 GPU, Intel Xeon Platinum 8163 CPU, 16GB RAM, running Ubuntu OS.

TABLE IV: The detailed statistics of the dataset

PROJ	THD	DUR(min)	PCP	MSG(*)	MSGLEN(*)
Angular	989	34	2.8	19.7(11.4)	12.2(20.9)
Spring-boot	985	145	2.3	9.3(6.0)	19.5(30.2)
Deeplearning4j	985	28	2.4	16.4(8.8)	12.0(22.4)

(*) denotes the statistics after merging consecutive messages from the same developer.

Testing Scenarios. The evaluations are performed under the following two scenarios:

1) *Five-fold-cross-validation for intra-project scenario.* In this scenario, annotated threads from chatrooms of the testing projects are partially available in the training set, meaning the model can gain knowledge of the testing project during the training. We conduct a stratified five-fold-cross-validation for each chatroom in this testing scenario. Specifically, threads are divided into five folds using stratified random sampling, with each fold preserving the original distribution of information types. Every time, we use four folds to train the model and the remaining one for testing. The process is repeated five times to alleviate the randomness, and we report the average evaluation results.
2) *Leave-one-project-out-cross-validation for cross-project scenario.* This scenario simulates the situation when users want to apply the trained algorithm to a new chatroom, which requires the learned knowledge of the model to be generalizable as the testing project is previously unseen during the training. We use a leave-one-project-out strategy, i.e., two projects are used as the *source projects* to train the model and the remaining one as the *target project* for testing. We iterate the process three times and report the average evaluation results.

Implementation Details. We use the pretrained BERT-Small model [45] from HuggingFace Transformer library [46] due to the limited computation resources. It produces a 512-dimensional embedding to represent each input message. The output dimension of Bi-LSTM is 512 (256 for each). Further, through a non-linear projection header, we get a 256-dimensional embedding for the textual features of the thread. Non-textual features are normalized before encoding. The dimension of the non-textual feature embedding is the same as the textual counterpart. The final representation of the thread in stage two, i.e., the concatenation of embeddings of two types of features, is a 512-dimensional embedding. We use cross-entropy as the loss function in the two stages.

To avoid the over-fitting problem, we apply dropout [47] to the outputs of every fully connected layer with the drop rate set to 0.1. We use AdamW [48] as the optimizer for model training in both two stages. In stage one, we set the learning rate (lr) to 1e-3 except for the pretrained BERT module, whose lr is set to 2e-5 as suggested in [39]. Specially, the pairs of threads used for training are sampled from the dataset in a 3:1 ratio of negative pairs (two threads with different classes) to the positive ones (two threads with the same class), which is considered as an optimal ratio for training siamese networks [49]. In stage two, we set lr to 1e-5 for fine-tuning the pretrained textual feature encoder and 1e-3 for other modules. Besides, we adjust class weights in the loss function to tackle the unbalanced dataset problem (Figure 2).

Evaluation Metrics. We use the following metrics to evaluate the performance of F2Chat: 1) *Precision.* Precision for class C_i is the ratio of the number of threads that are correctly classified as C_i to the total number of predictions made for C_i. 2) *Recall.* Recall for class C_i is the ratio of number of threads that are correctly classified as C_i to the total number of threads that belong to C_i in the ground truth. 3) *F1-score.* F1-score for class C_i is the harmonic mean of its precision and recall. The above three metrics evaluate the performance for a specific category. For the evaluation of the overall performance, we calculate the average *F1-score* of all classes weighted by *Support*, i.e., the number of threads of each class in the test set. These metrics are widely adopted in the previous studies that involve classification of software artifacts [13], [20], [21].

B. Research Questions

Our evaluation explores the following research questions:

RQ1: Does our approach work well in the intra-project scenario? We follow the five-fold-cross-validation described in Section V-A to evaluate the performance of our approach in the intra-project scenario. We use FRMiner, proposed by Shi *et al.* [12], as our baseline. To the best of our knowledge, FRMiner is the only one targeted for thread-level classification of developer communication artifacts. The experimental results suggest that FRMiner substantially outperforms two advanced sentence-level approaches [21], [50] and four general text classification approaches [51]–[54]. Furthermore, F2CHAT-t (stage one of our approach) uses the same siamese architecture as in their work to pretrain the textual feature encoder. Hence, we only consider FRMiner for performance comparisons with our approach in the experiments. However, FRMiner is designed to identify one certain type of discussion threads in developer chatrooms, i.e., threads with hidden feature requests. To enable a comparison, we adapt it to our task by modifying the number of output units in the last classification layers and retraining on our dataset using the same settings in [12].

RQ2: Does our approach work well in the cross-project scenario? To evaluate the performance of our approach in the cross-project scenario, we follow the leave-one-project-out-cross-validation described in Section V-A. We also use FRMiner as our baseline for this RQ.

RQ3: How does our approach benefit from the handcrafted non-textual features? We investigate whether our identified non-textual features help. To do so, we compare the performance of F2CHAT-t and F2CHAT (leverages two types of features) in both intra-project and cross-project scenarios.

TABLE V: The performance comparisons between our approach and FRMiner for each chatroom in intra-project setting

Metric	Approach	Chatroom			Avg.
		Angular	Spring-boot	Deeplearning4j	
Precision	FRMiner	0.384	0.425	0.404	0.404
	F2CHAT-t	0.659	0.576	0.543	0.593
	F2CHAT	**0.686**	**0.647**	**0.588**	**0.640**
Recall	FRMiner	0.444	0.460	0.407	0.437
	F2CHAT-t	0.663	0.610	0.555	0.609
	F2CHAT	**0.689**	**0.650**	**0.593**	**0.644**
F1-score	FRMiner	0.398	0.403	0.399	0.400
	F2CHAT-t	0.656	0.581	0.519	0.585
	F2CHAT	**0.681**	**0.632**	**0.572**	**0.628**

C. Experiment Results

RQ1: Performance in Intra-project Validation. The performance comparisons for each chatroom under the intra-project setting are shown in Table V. Here, the precision, recall, and F1-score are averages of all information categories weighted by support. The best results are highlighted in bold. F2CHAT-t improves the performance of FRMiner (w.r.t. F1-score) by 64.8%, 44.2%, and 30.1% for each of the three chatrooms, respectively. Considering both approaches using only textual features and sharing the same siamese architecture, the performance improvement verifies the effectiveness of 1) Better preprocessing of chat messages (Section IV-A) and 2) Introducing BERT for sentence modeling (Section IV-B). Although F2CHAT-t has already achieved satisfactory results by leveraging deep textual features, F2CHAT further boosts the performances in all three chatrooms, with an average of 0.640, 0.644, and 0.628 in precision, recall, and F1-score.

TABLE VI: The average performance of F2CHAT across three chatrooms for each information type in intra-project setting

Information Category	Precision	Recall	F1-score	Support
Programming Problem (PP)	0.638	0.690	0.663	718
Library Problem (LP)	0.620	0.577	0.581	303
Documentation Problem (DP)	0.333	0.183	0.236	75
Programming Information (PI)	0.749	0.834	0.787	1,145
Library Information (LI)	0.453	0.541	0.474	181
Documentation Information (DI)	0.472	0.446	0.455	176
General Information (GI)	0.250	0.107	0.150	161
Technical Discussion (TD)	0.815	0.222	0.318	117
Task Progress (TP)	0.467	0.241	0.318	83
Weighted Avg.	0.640	0.644	0.628	—

The average performance achieved by F2CHAT across all three chatrooms for each information category is shown in Table VI. Generally, F2CHAT achieves better performance on information categories with larger support. The best performance is on *Programming Information* with an average F1-score of 0.787. For categories where F2CHAT performs poorly, it is mainly due to the limited data samples, thus failing to capture effective patterns. The performance comparisons for each information category are shown in Table VII. Here, the results are averaged across all three chatrooms. The best results are highlighted in bold. Through manual checking of specific testing samples, we find that compared with FRMiner, F2CHAT-t can better distinguish between *Programming Problem* and *Programming Information*, as well as *Library Problem* and *Library Information*. It benefits from fine-grained special tokens replacement (Section IV-A). Since source code and stack trace embedded in the message text share the same format (both of them are considered as code snippets), replacing them with the same tag will confuse the learning-based model. Besides, by incorporating non-textual features, F2CHAT further improves the performance on almost every information category.

TABLE VII: The performance comparisons for each information type in intra-project setting

Metric	Approach	Information Category								
		PP	LP	DP	PI	LI	DI	GI	TD	TP
Precision	FRMiner	0.339	0.429	0.051	0.574	0.236	0.240	0.215	0.111	0.167
	F2CHAT-t	0.622	0.561	0.111	0.720	0.377	0.229	0.210	0.622	**0.500**
	F2CHAT	**0.638**	**0.620**	**0.333**	**0.749**	**0.453**	**0.472**	**0.250**	**0.815**	0.467
Recall	FRMiner	0.235	0.376	0.067	0.718	0.359	0.304	0.126	0.139	0.056
	F2CHAT-t	0.592	0.453	0.100	**0.870**	0.489	0.325	**0.137**	**0.278**	0.130
	F2CHAT	**0.690**	**0.577**	**0.183**	0.834	**0.541**	**0.446**	0.107	0.222	**0.241**
F1-score	FRMiner	0.254	0.377	0.058	0.634	0.282	0.256	0.149	0.123	0.083
	F2CHAT-t	0.603	0.501	0.105	0.784	0.426	0.258	0.145	**0.346**	0.198
	F2CHAT	**0.663**	**0.581**	**0.236**	**0.787**	**0.474**	**0.455**	**0.150**	0.318	**0.318**

RQ2: Performance in Cross-project Validation. The performance comparisons for each chatroom under the cross-project setting are listed in Table VIII. Here, the chatroom refers to the one selected as the *target project* (Section V-A). In general, the findings are similar to those in the intra-project setting. F2CHAT-t surpasses the FRMiner, and F2CHAT further boosts the performance. Compared with the results in the intra-project

TABLE VIII: The performance comparisons for each chatroom in cross-project setting

Metric	Approach	Chatroom			Avg.
		Angular	Spring-boot	Deeplearning4j	
Precision	FRMiner	0.435	0.485	0.393	0.438
	F2CHAT-t	0.601	0.539	0.482	0.541
	F2CHAT	**0.646**	**0.604**	**0.530**	**0.593**
Recall	FRMiner	0.438	0.468	0.395	0.434
	F2CHAT-t	0.573	0.530	0.515	0.539
	F2CHAT	**0.631**	**0.577**	**0.542**	**0.583**
F1-score	FRMiner	0.419	0.464	0.328	0.404
	F2CHAT-t	0.572	0.522	0.465	0.520
	F2CHAT	**0.616**	**0.584**	**0.517**	**0.572**

validation (Table V), the performance of F2CHAT declines on all three chatrooms by 8.9% over the average F1-score. However, the performance changes of FRMiner vary across different chatrooms. We observe an improvement on *Angular* and *Spring-Boot*, while a decline for *Deeplearning4j*. The improvement is mainly due to a larger training set. In the cross-project setting, all threads from two chatrooms (leave-one-project-out-cross-validation) are available for training, while in the intra-project setting, only 80% of threads from a single chatroom (five-fold-cross-validation) are available. The huge decline on *Deeplearning4j* is because it is a library for supporting deep learning algorithms, but the other two projects (*Angular* & *Spring-Boot*) are related to web development, thus making the linguistic patterns learned during the training hard to generalize on the testing set.

TABLE IX: The performance comparisons for each information type in cross-project setting

Metric	Approach	Information Category								
		PP	LP	DP	PI	LI	DI	GI	TD	TP
Pre-cision	FRMiner	0.487	0.352	0.000	0.596	0.238	0.224	0.143	0.056	0.367
	F2CHAT-t	**0.656**	0.480	**0.333**	0.685	0.321	0.194	0.195	0.172	0.194
	F2CHAT	0.613	**0.593**	0.204	**0.729**	**0.464**	**0.451**	**0.204**	**0.238**	**0.688**
Re-call	FRMiner	0.449	0.420	0.000	0.610	0.251	0.259	0.178	0.095	0.058
	F2CHAT-t	0.575	**0.517**	0.033	**0.772**	**0.466**	0.120	0.132	0.228	0.057
	F2CHAT	**0.695**	0.459	**0.070**	0.751	0.330	**0.409**	**0.154**	**0.302**	**0.315**
F1-score	FRMiner	0.393	0.383	0.000	0.582	0.221	0.239	0.116	0.070	0.099
	F2CHAT-t	0.610	0.495	0.061	0.723	0.331	0.148	0.054	0.153	0.088
	F2CHAT	**0.651**	**0.508**	**0.093**	**0.738**	**0.372**	**0.337**	**0.146**	**0.261**	**0.419**

The average performance achieved by FRMiner and our approach across the three chatrooms for each information category is shown in Table IX. The results further verify the effectiveness of F2CHAT under the cross-project setting as it outperforms the FRMiner on every information category. Compared with the results under the intra-project setting (Table VI), larger declines are observed on information types with smaller support. Besides, we find the performance of both FRMiner and F2CHAT-t on *Programming Problem* and *Library Problem* are close to or even better than the results in intra-project validation. This suggests that threads of these two categories share the most similar linguistic patterns that are irrelevant to project-specific concepts.

RQ3: Effectiveness of Handcrafted Non-textual Features. To evaluate the effectiveness of our identified non-textual features, we focus on the comparison of F2CHAT with F2CHAT-t in this RQ. First, we discuss the effectiveness of handcrafted non-textual features under the intra-project setting. As shown in Table V, by incorporating non-textual features, F2CHAT outperforms F2CHAT-t in all three chatrooms, indicating our identified features (Table III) can supplement the textual counterpart with effective information (e.g., thread structure, discussion participants). Besides, we observe that non-textual features are more powerful when the performances are poorer using only textual features. F2CHAT improves the F1-score of F2CHAT-t by 10.2% for *Deeplearning4j*, while only 3.8% for *Angular*. Regarding specific information categories, non-textual features are more effective on certain types (Table VII). This indicates that non-textual features, such as the duration of the thread, number of participants, and number of special tokens (e.g., URL, issue), are strong indicators of information categories, including *Library Problem*, *Documentation Information* and *Task Progress*. While for *General Information* and *Technical Discussion*, adding non-textual features does not help or even decreases the performance. This suggests that threads of these categories vary a lot in lengths, structures, and discussion contents.

Second, we discuss the effectiveness of handcrafted non-textual features under the cross-project setting. The results presented in Table VIII suggest that our identified non-textual features are generalizable across different projects, improving the average F1-score by 10.0% in cross-project setting compared with 7.4% under intra-project setting (Table V). When investigating the generalizability of non-textual features on specific information categories, we observe that they are more generalizable on *Documentation Information* and *Task Progress* (Table IX). Although the performance on these categories using only textual features is relatively poorer, as we discussed in the last paragraph that this could be the cause for larger improvements, it reveals that discussions of these categories share the most similar structures and formats even in different projects and communities.

VI. DISCUSSION

Effectiveness of Two-stage Model Design. We discuss the motivations for separating the training process of textual and non-textual feature encoders in Section IV-B. Here, we conduct an ablation study to further verify the effectiveness of our two-stage model design. We replace the $Encoder_{text}$ in Figure 3(a) into the architecture in Figure 3(b) without the last classification layers (the feed-forward module and the softmax layer), and refer this model as F2CHAT-s. F2CHAT-s simultaneously train both encoders for textual and non-textual features using the siamese architecture. We compare its performance against F2CHAT-t and F2CHAT under intra-project scenario. F2CHAT-t shares the same siamese architecture, but it only leverages textual features. F2CHAT utilizes two types of features, but it separates the training processes. The average performances of each model across three chatrooms are shown in Table X. The performances of F2CHAT-s and F2CHAT-t are close, while F2CHAT significantly surpasses both two models.

TABLE X: The performance comparisons in ablation study

	Precision	Recall	F1-score
F2CHAT-s	0.592	0.604	0.582
F2CHAT-t	0.593	0.609	0.585
F2CHAT	0.640	0.644	0.628

Effectiveness of Incorporating Hand-crafted Non-textual Features. Although the deep textual features are powerful in presenting the semantic information, we argue that they only focus on the linguistic aspect, while neglecting other information of the discussion thread (e.g., structure, participant). Moreover, the textual features do not consider the differences brought by the speaker's identity (i.e., the asker and respondents). Based on the observations from manual analysis of the data, we discuss that these kinds of information can contribute to the prediction of information categories (Section IV-C). Hence, we supply F2CHAT with handcrafted non-textual features to bridge the information gap. Figure 5 shows examples to illustrate the effectiveness of non-textual features.

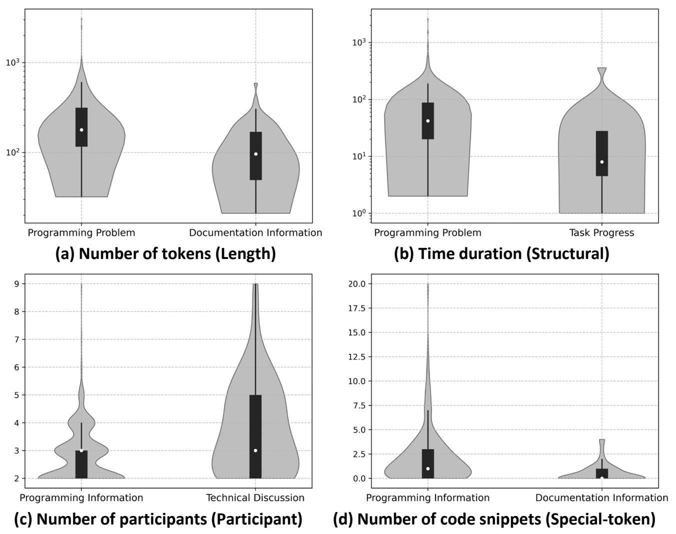


Fig. 5: Distributions of four non-textual features (one from each type) of different information types in Angular chatroom

VII. Threats to Validity

There are two major threats to the validity of this work. 1) The generalizability of the identified information categories and the proposed mining technique. We only sample 2,959 threads from three active chatrooms on Gitter due to the huge cost of manual analysis of developer discussion threads. To ensure the diversity of the sampled data, we select chatrooms with several principals (Section III-A), e.g., most actively used and belonging to different categories. However, the project-specific characteristics of the selected chatrooms might affect the generalizability of our identified information categories and the performance of the proposed automated mining technique. Specifically, in the preprocessing of chat messages (Section IV-A), we argue that the special tokens should be replaced based on the specific artifacts contained instead of their forms. We notice that developer communications, including instant chats, discussions in ITS, etc., typically involve various software artifacts (e.g., code snippets, issue reports and stack traces). These artifacts are closely related to the discussion topics, and are embedded in the text using diverse forms. Hence, we argue that the core idea of fine-grained special tokens replacement is applicable to other mining sources of developer communications. However, the detailed implementations and effectiveness may vary across different sources. 2) The quality of the dataset used for evaluation. We build a larger dataset (2,959 threads) than the prior work [12], which has only 1,035 threads and focuses on one specific information type. We leverage the state-of-the-art method proposed by Ehsan et al. [8] to disentangle the stream of messages to threads. However, the disentangled threads may contain more or fewer messages, confusing the thread-level classification techniques. Moreover, the ground-truth labels of the threads are manually annotated, which may subject to the personal experience of the annotators. Two annotators work collaboratively following a card-sorting process to eliminate this inconsistency.

VIII. Conclusion and Future Work

In this paper, we conduct an in-depth analysis to identify the information categories available in discussion threads that may satisfy the diverse needs of various OSS stakeholders. First, we build a thread-level taxonomy with nine information categories through manual examination of 2,959 threads from three chatrooms on Gitter. Second, we propose an automated classification technique, namely F2CHAT, which combines handcrafted non-textual features with deep textual features extracted by neural models. Evaluation results suggest that F2CHAT outperforms FRMiner by 57% with an average F1-score of 0.628. Our approach also achieves considerable performance in cross-project validation, which indicates F2CHAT can extract patterns that are generalizable across various projects. The experiment results also verify the effectiveness of our indentified non-textual features under both intra-project and cross-project validation. In future work, we plan to refine our taxonomy by exploring discussions from more chatrooms. We also plan to develop a tool for Gitter chatrooms to support the daily development tasks of OSS stakeholders by providing well structured and categorized historical chat data.

IX. Acknowledgments

This research/project is supported by the National Science Foundation of China (No. U20A20173 and No. 6190234), Key Research and Development Program of Zhejiang Province (No.2021C01014), and National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] B. Lin, A. Zagalsky, M.-A. Storey, and A. Serebrenik, "Why developers are slacking off: Understanding how software teams use slack," in *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, 2016, pp. 333–336.

[2] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky, "The (r) evolution of social media in software engineering," in *Future of Software Engineering Proceedings*, 2014, pp. 100–116.

[3] V. Käfer, D. Graziotin, I. Bogicevic, S. Wagner, and J. Ramadani, "Communication in open-source projects-end of the e-mail era?" in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 242–243.

[4] "Gitter." [Online]. Available: https://gitter.im/

[5] "Slack." [Online]. Available: https://slack.com/

[6] "Discord." [Online]. Available: https://discord.com/

[7] H. Sahar, A. Hindle, and C.-P. Bezemer, "How are issue reports discussed in gitter chat rooms?" *Journal of Systems and Software*, vol. 172, p. 110852, 2019.

[8] O. Ehsan, S. Hassan, M. E. Mezouar, and Y. Zou, "An empirical study of developer discussions in the gitter platform," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–39, 2020.

[9] P. Chatterjee, K. Damevski, L. Pollock, V. Augustine, and N. A. Kraft, "Exploratory study of slack q&a chats as a mining source for software engineering tools," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 490–501.

[10] R. Alkadhi, T. Lata, E. Guzmany, and B. Bruegge, "Rationale in development chat messages: an exploratory study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 436–446.

[11] R. Alkadhi, M. Nonnenmacher, E. Guzman, and B. Bruegge, "How do developers discuss rationale?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 357–369.

[12] L. Shi, M. Xing, M. Li, Y. Wang, S. Li, and Q. Wang, "Detection of hidden feature requests from massive chat messages via deep siamese network," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 641–653.

[13] D. Arya, W. Wang, J. L. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 454–464.

[14] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 12–23.

[15] 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 *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 281–290.

[16] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 375–385.

[17] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 767–778.

[18] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *2010 ACM/IEEE 32nd International conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 505–514.

[19] P. Rodeghero, S. Jiang, A. Armaly, and C. McMillan, "Detecting user story information in developer-client conversations to generate extractive summaries," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 49–59.

[20] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 491–502.

[21] Q. Huang, X. Xia, D. Lo, and G. C. Murphy, "Automating intention mining," *IEEE Transactions on Software Engineering*, 2018.

[22] "Our replication package." [Online]. Available: https://github.com/panshengyi/F2Chat

[23] E. Shihab, Z. M. Jiang, and A. E. Hassan, "Studying the use of developer irc meetings in open source projects," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 147–156.

[24] ——, "On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 107–110.

[25] P. Chatterjee, K. Damevski, N. A. Kraft, and L. Pollock, "Software-related slack chats with disentangled conversations," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 588–592.

[26] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[27] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. [Online]. Available: https://www.aclweb.org/anthology/D14-1181

[28] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–34, 2020.

[29] "Angular chatroom on gitter." [Online]. Available: https://gitter.im/angular/angular

[30] "Spring-boot chatroom on gitter." [Online]. Available: https://gitter.im/spring-projects/spring-boot

[31] "Deeplearning4j chatroom on gitter." [Online]. Available: https://gitter.im/eclipse/deeplearning4j

[32] "Gitter explore page." [Online]. Available: https://gitter.im/home/explore

[33] "Gitter developer page." [Online]. Available: https://developer.gitter.im/

[34] M. Elsner and E. Charniak, "You talking to me? a corpus and algorithm for conversation disentanglement," in *Proceedings of ACL-08: HLT*, 2008, pp. 834–842.

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

[36] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[37] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[38] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, pp. 737–737, 1994.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[40] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[41] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, pp. 1–26, 2020.

[42] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[43] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[44] G. Murray and G. Carenini, "Summarizing spoken and written conversations," in *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, 2008, pp. 773–782.

[45] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: On the importance of pre-training compact models," *arXiv preprint arXiv:1908.08962v2*, 2019.

[46] "Bert-small on huggingface." [Online]. Available: https://huggingface.co/google/bert_uncased_L-4_H-256_A-4

[47] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[48] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.

[49] P. Neculoiu, M. Versteegh, and M. Rotaru, “Learning text similarity with siamese recurrent networks,” in *Proceedings of the 1st Workshop on Representation Learning for NLP*, 2016, pp. 148–157.

[50] L. Shi, C. Chen, Q. Wang, S. Li, and B. Boehm, “Understanding feature requests by leveraging fuzzy method and linguistic analysis,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 440–450.

[51] A. McCallum, K. Nigam *et al.*, “A comparison of event models for naive bayes text classification,” in *AAAI-98 workshop on learning for text categorization*, vol. 752, no. 1. Citeseer, 1998, pp. 41–48.

[52] A. Liaw, M. Wiener *et al.*, “Classification and regression by randomforest,” *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[53] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “Lightgbm: A highly efficient gradient boosting decision tree,” *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[54] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, “Fasttext. zip: Compressing text classification models,” *arXiv preprint arXiv:1612.03651*, 2016.