Requirements Engineering (2021) 26:255–271
https://doi.org/10.1007/s00766-020-00344-y

ORIGINAL ARTICLE

Automatically detecting feature requests from development emails by leveraging semantic sequence mining

Lin Shi[1,5] · Celia Chen[2] · Qing Wang[1,3,5] · Barry Boehm[4]

Received: 12 January 2020 / Accepted: 16 November 2020 / Published online: 30 March 2021

Abstract
Mailing list is widely used as an important channel for communications between developers and stakeholders. It consists of emails that are posted for various purposes, such as reporting problems, seeking help in usage, managing projects, and discussing new features. Due to the intensive amount of new incoming emails every day, some valuable emails that intend to describe new features may get overlooked by developers. However, identifying these feature requests from development emails is a labor-intensive and challenging task. In this paper, we propose an automated solution to discover feature requests from development emails by leveraging semantic sequence patterns. First, we tag sentences in emails by using 81 fuzzy rules proposed in our previous study. Then we represent the semantic sequence with the contextual information of an email in a 2-g model. After applying sequence pattern mining techniques, we generate 10 semantic sequence patterns from 317 tagged emails that are randomly sampled from the Ubuntu community. We also conduct an empirical evaluation of their capability to discover feature requests from massive emails in Ubuntu and other four open source communities. The results show that our approach can effectively identify feature requests from these emails. Compared to existing baselines, our approach can achieve a better performance in terms of precision, recall, F1-score, AUC, and positive, with the average precision and recall for discovering feature requests from emails being 76% and 86%, respectively.

Keywords Requirements discovery · Requirements analysis · Text mining · Feature requests

1 Introduction

Obtaining a sufficient number of requirements is crucial in software development as it increases the opportunity to gain market share and secure more customers. To achieve that goal, traditional RE approaches typically select a limited number of stakeholders and crowd representatives to collect user requirements [21]. In the past decades, there has been a massive increase in global collaboration via online platforms, such as Github and JIRA. The large number of stakeholders makes the traditional activities of requirements gathering and analyzing extremely costly and time-consuming, and thus, these approaches miss the opportunity to continuously involve large groups of users who express their feedback or feature requests through a variety of media. The new trend is now shifting toward the CrowdRE, which focuses on automation or semi-automation of the requirements gathering process so that validated user requirements can be derived from a crowd [22].

Currently, mailing-lists act as one of the most frequently used communication channels that enable users to easily

✉ Qing Wang
wq@iscas.ac.cn

Lin Shi
shilin@iscas.ac.cn

Celia Chen
qchen2@oxy.edu

Barry Boehm
boehm@usc.edu

[1] Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences, Beijing, China

[2] Department of Computer Science, Occidental College, Los Angeles, USA

[3] State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences, Beijing, China

[4] Center for Systems and Software Engineering, University of Southern California, Los Angeles, USA

[5] University of Chinese Academy of Sciences, Beijing, China

submit feature requests and greatly improve the efficiency for organizations when gathering new ideas. For example, in the Ubuntu community, there are 24 among 358 mailing lists that are designated for user feedback [16]. However, there are challenges to gather and analyze these feature requests.

First, the volume of emails can be very large, and it is easy for developers to miss any requested features when they get lost in other unnecessary contents [54]. For example, apart from other active communication channels, the Ubuntu community has around 4000 daily incoming emails from its mailing-lists [16]. We randomly selected 507 emails from one of the Ubuntu mailing-lists for users and found that 116 of them were feature requests, but only a few of them were recorded and traced in the issue tracking system. Developers contribute to the open source communities under the burden of daily programming tasks as well as a massive amount of incoming emails. In such a situation, emails that are requesting new features are likely to be ignored, which has been confirmed by existing research. Guzzi et al. [23] reported that core developers participate in less than 75% of the threads in mailing-lists, and only 54% of emails suggesting features get processed.

The second challenge is that emails in the mailing-lists may relate to a variety of topics. For example, some emails may be posted for user complaints, bugs, or feature requests [7], while some emails may be posted for opinion asking or information seeking [52]. Moreover, Herzig et al. [25] and Antoniol et al. [5] found that over 30% of all user feedback are misclassified in issue tracking systems (i.e., rather than referring to a new feature, they resulted in an update of documentation, or a code fix). Hence, defining an enforced rule that any email proposing a feature must include a tag 'Feature Request' might also not help, but result in a number of invalid user requirements. To identify user requirements from emails, developers have to read through all the emails carefully and separate feature requesting emails from other emails, which will inevitably increase their workload.

In this paper, we propose an automated solution to discover feature requests from massive textual emails. First, we classify sentences in the emails into six categories: *Intent, Benefit, Drawback, Example, Explanation*, and *Trivia*. These categories represent the semantic meanings by leveraging 81 fuzzy rules proposed by Shi et al. [50]. Second, we transform the emails into the semantic sequences based on the classification results. By mining the sequences, we identify semantic sequence patterns that can indicate feature requests from development emails. Ten semantic sequence patterns are reported, and we conduct an empirical evaluation toward their capability to discover feature requests from massive emails in Ubuntu and other four open source communities. The results show that our approach can effectively identify feature requests from these emails. Compared to five existing baselines, our approach has a better performance in terms of precision, recall, F1-score, AUC, and positive.

The major contributions of this paper are as follows.

- We propose an automated solution to discover feature requests from a large volume of development emails by providing 10 semantic sequence patterns that can achieve satisfying performance under different business objectives.
- We conduct an empirical evaluation on Ubuntu and four other open source communities to discover feature requests from a large volume of development emails. The results conform to the generalizability and usability of the proposed approach.
- We provide publicly available tools (FRAD) and dataset to replicate our experiments.[1]

The remainder of the paper is organized as follows. Section 2 elaborates on the approach. Section 3 presents the experimental setup. Section 4 describes the results and analysis. Section 5 discusses the implications and future work. Section 6 shows the threats to validity. Section 7 introduces the related work. Section 8 concludes our work.

2 Approach

Our approach is inspired by Shi et al.'s research [49] about automated analysis on the contents of feature requests. They proposed 81 fuzzy rules that can classify textual sentences into six semantic tags (i.e., Intent, Benefit, Drawback, Example, Explanation, and Trivia). By providing the tagged contents, one can understand and analyze the requirements in an efficient way. Our work takes advantage of the tagged contents, and mines the sequences of sentence tags to learn common expressing patterns in feature requests, which can be used to identify feature requests from any textual mediums, such as Github and online reviews.

The process of our approach is illustrated in Fig. 1. First, we tag sentences in the emails into six tags according to the 81 fuzzy rules proposed by Shi et al. [49]. After that, we generate the corresponding semantic sequences for emails. Then we apply sequence patterns mining algorithm to discover candidate feature request semantic patterns. We use Fn-score as the optimization function to output the final semantic sequence patterns, which can be used to distinguish feature requests emails from other types of emails.

[1] http://39.104.76.212:8082.

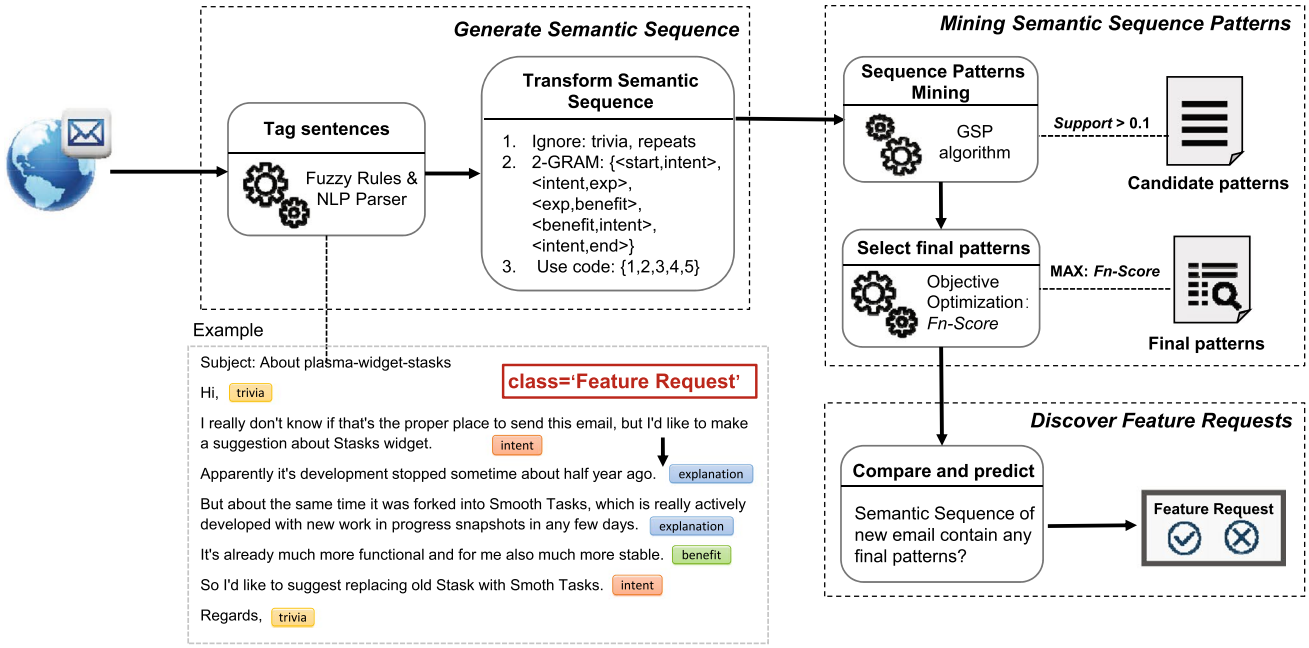

Fig. 1 The approach overview

Table 1 Definitions of sentence categories

Category	Importance	Definition
Intent	1	Descriptions about ideas, needs, or expectations to improve the system and its functionalities
Benefit	2	Descriptions about good or helpful results or effects that the proposed feature will deliver
Drawback	3	Descriptions of disadvantages or the negative parts of the current system behavior
Example	4	Descriptions of examples or references in support of the proposed feature
Explanation	5	Detailed information about the current behavior, scenarios, or solutions related to the proposed feature
Trivia	6	Other information that are not related to the proposed feature nor the system

2.1 Generate semantic sequences

To generate the semantic sequence for each email, we first classify sentences in the emails into six semantic categories based on Shi et al.'s previous work [49]. The six categories are defined in Table 1. We utilize the API service provided by Shi et al., which automatically returns the tagging results for a given text. Second, we generate concise semantic sequences and transform them into 2-g semantic sequences. Finally, we transform the 2-g semantic sequences into natural numbers for sequence mining.

Given an email with tags, we can obtain all the sentence categories in the order of the expressing sequence. Taking the email in Fig. 1 as an example, we can obtain its expressed sequence: {*trivia, intent, explanation, explanation, benefit, intent, trivia*}.

To keep the sequence concise, we adjust the original expressing sequence with the following steps:

- Excluding sentences with 'trivia' tags.
- Combining the same consecutively repeating tags.
- Adding Start(S) and End(E) tags.

Given n sentences, we generate the simplified semantic sequences as follows:

$$Q = \{S, t_1, t_2, \ldots, t_n, E\} \tag{1}$$

where t_i is the semantic category of sentence i, S denotes the starting of an email, and E denotes the ending. In this work, the number of semantic categories is 5 after ignoring the trivia category. Therefore, after adjusting the original expressing sequence, the concise semantic sequence of the given example is *S, intent, explanation, benefit, intent, E.*

In order to capture the contextual information of each sentence in the email, we leverage the *N-gram* model to represent the semantic sequences of each sentence. *N-gram* model is a contiguous sequence of n items from a given sample of

text, which is widely used in natural language processing [46] and biological sequence analysis [13] . In this study, we use *2-g* model to represent the concise semantic sequences because of the following two reasons: (1) it records more context information and can retain the relationship between neighbor semantic categories [12]; (2) it enables the experiments to scale up efficiently [42].

The 2-g model of Q is defined as follows.

$$\text{Bigram}(Q) = \{\langle S, t_1\rangle, \langle t_1, t_2\rangle, \langle t_2, t_3\rangle, \ldots, \langle t_n, E\rangle\} \quad (2)$$

To perform sequence mining on the 2-g model, we further transform the sequence of tag-pairs into the sequence of natural numbers. Since we combine the same consecutively tags, the two tags appear in one pair will not be identical. Thus, given m different categories used in semantic sequence analysis, there will be P_m^2, which is $(m^2 - m)$ pairs of $\langle t_i, t_j\rangle$. After adding m pairs that start with S and m pairs that end with E, the total number of pairs is $P_m^2 + 2 \times m$, which is $m \times (m + 1)$.

In this work, there are 5 different tags ($m = 5$), and there are 30 different pairs, which means that there will be 30 available combinations as units in the transformed 2-g semantic sequences.

To simplify the data presentation and apply the sequence mining algorithm, we use the natural number of the 2-g model combinations to represent the semantic sequence. Taking the semantic sequence Q_ψ ={S, intent, expression, benefit, intent, E} in Fig. 1 as an example, the 2-g model $Bigram(Q_\psi)$ is {<S, intent >, <intent, expression>, <expression, benefit>, <benefit, intent>, <intent, E>}. The order of <S, intent >e is 1 among the 30 available combinations, the order of <intent, expression> is 9, and so on. Therefore, the semantic sequence of Q_ψ is {1, 9, 23, 10, 26}.

2.2 Mining semantic sequence patterns

The semantic sequences of development emails consist of the numbers that denote the tag-pairs. The sequence of the numbers could reflect the expressing logic in the emails. Finding the frequently occurring sequential patterns in the semantic sequence of feature-request emails can reveal knowledge about common expressing logic when people describe desired features.

Since sequential pattern mining is a topic of data mining that concerns with finding statistically relevant patterns [51], we can extract frequent patterns by applying sequential pattern mining on the semantic sequences of development emails. Many sequential pattern mining algorithms have been proposed [40, 53, 56]. In our study, we select the widely used apriori-based algorithm: Generalized Sequential Pattern (GSP) algorithm [53].

The process of mining semantic sequence patterns is presented in Algorithm 1. We break it down into two steps. First, we apply the GSP sequential pattern mining algorithm on the emails that are labeled as feature requests to find frequent semantic sequences. Each frequent semantic sequence comes with its corresponding *support* value and *confidence* value. The *support* and *confidence* measurements are typically used in data mining to evaluate rule-based classifiers [2]. In our study, we use *support* and *confidence* to help us select semantic sequence patterns. We select those frequent semantic sequences with *support* over 0.1 as the semantic sequence patterns candidates. Second, we calculate the *confidence* of each semantic sequence pattern candidate among all the dataset, and rank the candidates by the values of *confidence*. We use Fn-score as the optimization function to output the final semantic sequence patterns for the i^{th} dataset. F1-score is also known as the harmonic mean of the *precision* and *recall* [41]. The 'Fn-score_HP' in Eq. (3) is defined for 'High Precision' objective, which means that the higher value of n, the higher weight of precision. For 'High Recall' objective, we simply exchange the position of precision and recall as shown in the definition of 'Fn-score_HR,' and increase the weight of recall by enlarging the value of n.

$$\begin{aligned}\text{Fn-score\_HP} &= \frac{(n+1) \times \text{Precision} \times \text{Recall}}{\text{Precision} + n \times \text{Recall}} \\ \text{Fn-score\_HR} &= \frac{(n+1) \times \text{Recall} \times \text{Precision}}{\text{Recall} + n \times \text{Precision}}\end{aligned} \quad (3)$$

Patterns are selected as final patterns only if the Fn-score does not decline when they are included. By tuning the value of n, we can generate optimal semantic sequence patterns for different business objectives. For example, patterns generated by $n = 10$ will output rules that aim to be more precise than the patterns generated by $n = 1$.

Algorithm 1 Semantic Sequence Patterns Mining

procedure SEMANTIC SEQUENCE PATTERNS MINING
Data: $Train(n)$: list of the semantic sequences of emails in n samples
Result: $optSSPList$: list of the optimum semantic sequence patterns, which record the sequence-pattern tuples, i.e, $\{Q_s, support, confidence\}$
    $Q_{old} \leftarrow \emptyset$
    $i \leftarrow 1$
    $Q_{opt} \leftarrow optimizationPatterns(Train(i))$
    while $Q_{old} == \emptyset \parallel Q_{opt} \not\approx Q_{old}$
    $Q_{old} \leftarrow Q_{opt}$
    $i \leftarrow i + 1$
    $Q_{opt} \leftarrow optimizationPatterns(Train(i))$
end procedure
function OPTIMIZATIONPATTERNS(List Trainset)
    $optList \leftarrow \emptyset$
    $candidateList \leftarrow \{p \mid p \in GSPAlgorithm(Trainset \rightarrow class = 1) \cap p.support > 0.1\}$
    $candidateList \leftarrow rankByConfidence(Trainset)$
    $max\_fscore \leftarrow 0$
    for $p \in candidateList$ **do**
        $optList \leftarrow optList \cup p$
        $fscore \leftarrow fscore(optList, Trainset)$
        if $fscore \geq max\_fscore$ **then**
            $max\_fscore \leftarrow fscore$
        else
            $optList \leftarrow optList - p$
        end if
    end for
    return optList
end function

2.3 Discover feature requests from texts by semantic sequence patterns

In Sect. 2.2, we have generated a set of final semantic sequence patterns. When applying these patterns to discover feature requests from new incoming emails, the following steps need to be taken. First, based on their business objectives, users need to choose the corresponding pattern from the generated patterns. For example, if the business objective is 'The prediction results should be highly precise,' then the user should select the patterns with a higher weight of Fn-score_HP. If the business objective is 'The prediction results should recall more actual feature requests,' then the user should select the patterns with the higher weight of Fn-score_HR. Second, when a new email is coming, we automatically obtain the semantic sequence by applying the same data processing procedure. Third, we compare the semantic sequence of the new email with the specified pattern. The specified pattern is like the 'DNA' of feature requests. If the given semantic sequence includes the full sequence of the specified pattern, then the corresponding email is predicted to be a feature request. Finally, our approach outputs the prediction result as well as the probability.

To illustrate the application of the semantic sequence patterns, we take the daily work of a release team member Adam, who aims to monitor and analyze potential new requirements, as an example. Suppose Adam would like to search only a limited number of feature requests from emails due to his tight schedule. He chooses the semantic sequence pattern P5 ([1 26, 13 17 22]) with the highest precision objective to help him discover a more accurate prediction on feature request email. When analyzing the email in Fig. 1, we can find that its semantic sequence is $Q_\psi = \{1, 9, 23, 10, 26\}$ as explained in Sect. 2.1. Then we compare P5 with Q_ψ, and we notice that Q_ψ contains the sequence of [1 26], which is one of the feature-request semantic sequences defined in P5. Therefore, we consider the Q_ψ matches with P5, and we recommend the incoming email as a feature request to Adam for further analysis.

2.4 Tools support: feature request analyzer and detector (FRAD)

Based on the proposed approach, we implement an automatically Feature Request Analyzer and Detector (FRAD) online system that can identify feature requests from emails. Given a raw email and the selected pattern, FRAD will first automatically tag each sentence by using the API service provided by Shi et al. [49]. Second, FRAD generates the semantic sequence for the given email as illustrated in Sect. 2.1. Third, FRAD will match the semantic sequence with the selected pattern. Only if the semantic sequence contains the selected pattern, the given email will be predicted as a feature request. More details can be found on our project site: http://39.104.76.212:8082/.

3 Experimental setup

In order to evaluate the effectiveness of our approach, three research questions are proposed in Sect. 3.1. To answer our research questions, we select and preprocess a set of open source projects in Sects. 3.2 and 3.3. In Sect. 3.4, we introduce measurements that are designed to evaluate the performance of the semantic sequence patterns. Then, we describe a detailed experiment design in Sect. 3.5.

3.1 Research questions

In our study, we investigate the performance of our approach. Specifically, the experiment aims at addressing the following research questions:

RQ1 (Effectiveness) *Can the proposed approach effectively discover feature requests from emails in the Ubuntu community?* This research question aims at investigating the kind of the semantic sequence patterns that can be generated

from the Ubuntu training dataset, and to what level the corresponding performances of these patterns can achieve on Ubuntu testing dataset.

RQ2 (Generalizability) *Can the semantic sequence patterns mined from Ubuntu community work well on other projects?* In RQ1, we train the semantic sequence patterns from Ubuntu training dataset, and evaluate in the Ubuntu testing dataset. However, it is arguable whether the semantic sequence patterns trained from Ubuntu project can apply to other projects. This research question aims to alleviate that concern by applying the semantic sequence patterns to other open source projects, and analyzes the performances.

RQ3 (Advantage) *How can the proposed approach perform when compared to the state-of-the-art approaches in identifying feature requests from emails?* This research question aims at comparing the performances of generated patterns with the existing approaches in terms of precision, recall, f1-score, AUC, and positive.

3.2 Subject projects

Ubuntu is an open source operating system software based on the Debian architecture. It is one of the distribution systems of Linux. Ubuntu has been releasing updated versions nearly every six months since its initial release in 2004. It has a large community with lots of active contributors internationally; thus there are a large quantity emails that are created and exchanged in its mailing lists. Due to the long history with consistent releases and a large volume of emails, we take the mailing lists of the Ubuntu community as the training dataset. In order to evaluate the performances as well as examine the generalizability of our approach, we conduct cross-project validation on emails of four open source projects from both Apache and Eclipse communities: Activemq, Aspectj, HDFS, and Jetty. All the mailing list discussions among developers are archived on Ubuntu Mailing lists [16] since December 2006. In our study, we chose one of the mailing lists Ubuntu-devel-discuss [17] that is designated for communications between users and developers.

Table 2 Cross-project validation subjects

Project	Community	Mailing list	Emails	Samples	FRs
Activemq	Apache	Activemq-users[a]	2996	99	19
Aspectj	Eclipse	Aspectj-users[b]	3232	100	24
HDFS	Apache	Hadoop-hdfs-user[c]	2330	95	26
Jetty	Eclipse	Jetty-users[d]	1604	100	18

[a] http://mail-archives.apache.org/mod_mbox/activemqusers

[b] http://dev.eclipse.org/mhonarc/lists/aspectjusers

[c] http://mail-archives.apache.org/mod_mbox/hadoop-hdfs-user

[d] https://accounts.eclipse.org/mailing-list/jetty-users

3.3 Data preparation

3.3.1 Data filtering

Typically, messages in the mailing list are organized in the form of threads. Developers first launch a mailing list thread by posting a head email that is for discussion, and then other developers reply to the same thread to share ideas, information, or suggestions. When preparing data for feature request discovery, we first collect the head emails from mailing list threads. We collect 4204 raw head emails in the Ubuntu-devel-discuss mailing list threads from December 2006 to July 2017. Typically the head email contains new ideas, questions, or requests, and follows by a series of further discussions. We ignore those threads that started with replies to other threads. After threads selection, we end up with 3434 head emails as our input data.

To conduct cross-project validation, we also select four popular open source projects from both Apache and Eclipse communities: ActiveMQ, AspectJ, HDFS, and Jetty . We target the mailing lists for users, and collect head emails from threads posted from the project creation time to Dec 2017. We randomly sample 100 emails out for each project as the testing dataset, and manually exclude unreadable emails:

- Emails that are written in non-English languages;
- Most of the emails are code or stack traces;
- Low-quality emails such as emails with many typos and grammatical errors.

The details of selected emails are shown in Table 2, and the last column 'FRs' denotes the number of feature requests.

3.3.2 Sampling

As labeling emails into the feature-request class or non-feature-request class requires thoroughly reading the textual contents, it involves heavy human resources during the labeling activity. Limited by the labeling resources, we perform an **incremental iterative sampling strategy** to prepare the dataset from the 3434 head emails taken from the Ubuntu community.

The incremental iterative sampling strategy includes three steps: (1) We randomly sample x percent of total emails without replacement that can be labeled within a limited cost, and define as dataset S_i; (2) we mine the semantic sequence patterns P_i from the united dataset $\bigcup_{i=1} S_i$; (3) we compare the similarity between P_{i-1} and P_i. If the similarity is over 80%, then we consider the P_i is representative and stop the sampling process. Otherwise, we repeat the process from the first step.

The similarity of two sets of semantic sequences patterns Q_i and Q_j is the proportion of their intersection elements over the size of Q_j. Note that we define the intersection of Q_i and Q_j as the set of elements of Q_j that are the subset of elements in Q_i.

$$\text{Similarity}(Q_i, Q_j) = \frac{|\, Q_i \cap Q_j \,|}{|\, Q_j \,|} \tag{4}$$

In this study, we randomly sample 2% emails (around 70 emails) for each iteration according to our limited labeling resource. After 4 iterations, we obtain the representative patterns for the Ubuntu community. Moreover, to validate the patterns in a more comprehensive way, we build nearly the same amount of data for testing. As a result, we have 259 emails for training and 248 emails for testing as shown in Table 3.

3.3.3 Label ground-truth emails

We labeled emails that are used as the ground-truth dataset for method definition and performance evaluation. To guarantee the correctness of the labeling results, we built an inspection team, which consisted of two senior researchers with seven Ph.D. candidates and three senior developers. All of them either have done intensive research work with software development or have been actively contributing to open source projects. We divided the team into two groups. Each group consisted of a leader (senior researcher) and five members. The leader trained members on how to label and provided consultation during the process. The labeling results from the members were reviewed by the leader, while results from the leaders were reviewed by other leaders. We accepted and included emails to our dataset when the emails received full agreement among the groups. When an email received different labeling results, we hosted a discussion with all the 12 people to decide through voting. If the majority of people vote for a particular class (i.e., feature-request class or non-feature-request class), then we labeled the email with the class that was supported by the majority.

Table 3 Ubuntu train and test dataset

	Dataset	# Emails	# FRs	#Sentences
Train	1	65	20	625
	2	132	33	1327
	3	197	43	2039
	4	259	65	2717
Test	1	248	51	2497

3.4 Evaluation measurements

In order to evaluate whether the generated semantic sequence patterns can effectively identify feature requests from new incoming emails, we use five measurements to evaluate the prediction performance: precision, recall, F1-score, positive, and AUC.

Precision, recall, and F1-score are commonly used measurements for performance assessment in classification tasks [41]. Precision represents the proportion of items labeled as belonging to class C that indeed belong to C. Recall represents the proportion of items from class C was labeled as belonging to Class C. The F1-score is the weighted average of precision and recall.

$$\text{Positive} = \frac{\text{TP} + \text{FP}}{\text{TP} + \text{FN} + \text{TN} + \text{FP}} \tag{5}$$

Positive [47] is defined as the proportion of the items labeled as belonging to class C among all the labeled ones, where TP, FP, TN, and FN represent for true positive, false positive, true negative, and false negative, respectively. In our study, the positive measurement can reflect the proportion of emails that are predicted to be feature requests. When a further analysis of the prediction results is required, this measurement can indicate the effort of such an analysis on the prediction results.

Area under ROC curve (AUC) is the area of the two-dimensional graph in which false positive rate is plotted on the X axis and true positive rate is plotted on the Y axis [20]. AUC can avoid performance inflation when evaluating on imbalanced data. The AUC value varies between 0 and 1, and higher values indicate better performance.

3.5 Experiment design

This section describes the designs of the experiments in detail.

Experiment I (Effectiveness) In this experiment, we first obtain the semantic sequence patterns for different optimization objectives from the Ubuntu training dataset. Then, we conduct within-project validation for those obtained patterns on the testing dataset from the Ubuntu project. There are two types of predefined business objectives:

- Objectives for achieving high precision: According to Eq. (4), we gradually increase the weight n of Fn-score_HP from 1 to 10 to achieve high precision objectives.

- Objectives for achieving high recall: We gradually increase the weight n of Fn-score_HR from 2 to 10 to achieve high recall objectives.

For each optimization objective, we use the GSP algorithm to mine its corresponding stable semantic sequence patterns on the five prepared training datasets incrementally. For each distinct stable semantic sequence pattern, we apply it on the testing dataset with 248 emails from the Ubuntu project, and observe the five measurements for prediction performance.

Experiment II (Generalizability) To address RQ2, we conduct cross-project validation for the obtained semantic sequence patterns on other open source projects (activemq, aspectj, hdfs, and jetty). After preparing the 4 testing dataset from the open source projects, we apply the semantic sequence patterns obtained in RQ1 on each testing dataset to predict whether emails are feature requests or not, and observe the five measurements for prediction performance.

Experiment III (Advantage) To compare the obtained semantic sequence patterns with the existing approaches, we selected DECA [18], which is the state-of-the-art approach for analyzing development emails content. It is used to classify the sentences of emails into feature request, opinion asking, problem discovery, solution proposal, information seeking, and information giving by using linguistic rules. Since the dataset provided by DECA is processed into numeric vectors[2] rather than the original textual emails, we cannot apply our approach on DECA dataset directly. Therefore, we apply DECA to the five testing dataset (i.e., Ubuntu_test, activemq, aspectj, hdfs, and jetty). We use the java API provided by DECA[3] to annotate the emails, and defined the emails that contain sentences that are predicted to be 'feature request' as feature request emails.

We select four representative machine learning approaches including Naive Bayes, J48, Logistic Regression, and SVM [43] to build classifiers from the 259 training dataset, and reported performances on the testing dataset, including the Ubuntu testing dataset and the four testing dataset from activemq, aspectj, hdfs, and jetty. We processed the training and testing dataset by applying vector format, STOP word filter, and TF-IDF weights to represent emails [44].

Moreover, we implement two deep learning approaches, TextCNN [29] and TextRNN [32] models by using the Keras framework. To obtain better performance, we use the grid search [11] method to tune the hyperparameters. In the word embedding layer of the model, the dimension of the word vector is 50 and the input length is 200. For TextCNN model, we set 3 different convolution kernel sizes which are 3, 4, and 5, respectively. For TextRNN model, we use the LSTM layer and set the number of hidden layer neurons to 128. To prevent overfitting, we set the dropout to 0.5 drop rate. We use Adam as the optimizer and cross-entropy as the loss function. In addition, we set the batch size to 8, meaning that it takes 8 data samples per training. We also set epochs to 50 and patience to 10, meaning that the entire training process needs 50 epochs, but when the performance on the validation set did not improve for 10 epochs, the process will be stopped.

4 Results and analysis

This section reports the analysis of the results achieved aiming at answering our research questions.

4.1 Answering RQ1 (effectiveness)

By applying the proposed approach on the Ubuntu training dataset, we obtained 10 different semantic sequence patterns over 19 weights of F_n-score for different objective types. We define them as P1 to P10, respectively, where P1 is patterns for the regular F_1-score, P2–P5 are for high precision objectives, and P6 to P10 are for high recall objectives. Some semantic sequence patterns can meet multiple objectives at the same time from observing the 19 stable patterns. For example, P4 can meet the high precision objectives on weight between 5 and 8 as shown in Table 4. 'Trained Emails' column represents the number of emails used for training stable patterns. We can see that except P2, all the other patterns become stable at the 3rd Ubuntu training dataset with 197 emails, which means that the proposed approach can obtain stable patterns on the Ubuntu training dataset quickly. 'Semantic Sequence Patterns' column represents the semantic sequence patterns that are mined by GSP from the training dataset. We append the bigram format of the semantic sequence patterns in the last column for more information.

To assess the performances of the 10 patterns, we apply these patterns on the Ubuntu testing dataset, which contains 248 new emails. We group the performances into a high precision patterns group and a high recall patterns group as shown in Figs. 2 and 3. The bar graphs in Figs. 2 and 3 denote the number of elements in each pattern. We can see that as the weights increase, the number of elements in the high precision patterns becomes smaller, and the number of elements in the high recall patterns becomes larger. The highest precision pattern P5 has 2 elements, and the highest recall pattern P10 has 11 elements.

[2] https://www.ifi.uzh.ch/dam/jcr:00000000-14e5-028d-ffff-ffffaffc5e6c/ReplicationpackageDECA.zip.

[3] https://www.ifi.uzh.ch/dam/jcr:00000000-5b34-b3d9-0000-00004910bd8d/DECA_API.zip.

In Fig. 2, the values of F1-score sharply decline after P2, while the values of AUC remain stable, which only slightly decline from 70 to 60%. Since AUC measures the accuracy of a classifier on all classes, we consider that all the five patterns can reach a relatively good accuracy on classifying feature requests from emails. The values of *positive* decline from 18 to 1%, which means the proportion of positive predictions over the whole test dataset is reducing. The *precision* is gradually increasing from 59 to 100% (slightly decline at P4), while the values of *recall* reduce from 51 to 4%. Taking the five measurements into consideration, we can see that as the weights increase, the trained patterns become more focused on a certain type of expressing logic. They can only recall a small proportion of actual feature requests, but the predicted emails are very likely to be actual feature requests, while the accuracy level of both sides can remain stable.

Figure 3 shows the performances of patterns generated by high recall objectives. By raising the weights of recall, the *recall* of the generated patterns increases from 53 to 73%, and the *precision* reduces from 50% to 26%. The values of *AUC* remain stable. They only slightly decline from 73 to 64%, which indicates a relatively good accuracy on classifying feature requests from emails. Taking the five measurements into consideration, we can see that as the weights increase, the trained patterns are likely to include more types of expressing logics. Although the precision turns to lower values, the patterns can recall most of the real feature requests, while the accuracy level of both sides can remain stable.

Table 4 Details of the 10 semantic sequence patterns

Objective type	ID	Weight	Trained emails	Semantic sequence patterns	Semantic sequence patterns in bigram
High precision	P1	1	197	[1, 6, 10, 2]	{<S,intent>},{<intent,benefit>},{<benefit,intent>},{<S,benefit>}
	P2	2	259	[1, 6, 10, 2 30]	{<S,intent>},{<intent,benefit>},{<benefit,intent>}, {<S,benefit>,<exp,E>}
	P3	[3, 4]	197	[1 26, 1 17, 6, 10]	{<S,intent>,<intent,E>}, {<S,intent>,<drawback,exp>}, {<intent,benefit>},{<benefit,intent>}
	P4	[5, 8]	197	[1 26, 1 17 22, 10]	{<S,intent>,<intent,E>}, {<S,intent>,<drawback,exp>, <exp,intent>}, {<benefit,intent>}
	P5	[9, 10]	197	[1 26, 13 17 22]	{<S,intent>,<intent,E>}, {<S,intent>,<drawback,exp>, <exp,intent>}
High Recall	P6	2	197	[1, 6, 10, 18, 2]	{<S,intent>},{<intent,benefit>}, {<benefit,intent>}, {<example,intent>},{<S,benefit>}
	P7	3	197	[1, 6, 10, 18, 2, 9 24, 22 26]	{<S,intent>},{<intent,benefit>},{<benefit,intent>}, {<example, intent>},{<S,benefit>}, {<intent,exp>,<exp,drawback>}, {<exp,intent>,<intent,E>}
	P8	4	197	[1, 6, 10, 18, 2, 26, 8, 9 24, 17 22, 13]	{<S,intent>},{<intent,benefit>}, {<benefit,intent>}, {<example, intent>}, {<S,benefit>},{<intent,E>}, {<intent,example>}, {<intent,exp>,<exp,drawback>}, {<drawback,exp>,<exp, intent>}, {<benefit,exp>}
	P9	[5, 7]	197	[1, 6, 10, 18, 2, 26, 27, 8, 9 24, 17 22, 13]	{<S,intent>},{<intent,benefit>}, {<benefit,intent>}, {<example, intent>}, {<S,benefit>},{<intent,E>}, {<benefit,E>},{<intent, example>}, {<intent,exp>,<exp,drawback>}, {<drawback,exp>, <exp,intent>},{<benefit,exp>}
	P10	[8, 10]	197	[1, 6, 10, 18, 2, 26, 27, 8, 24, 17 22, 13]	{<S,intent>},{<intent,benefit>},{<benefit,intent>}, {<example, intent>},{<S,benefit>},{<intent,E>}, {<benefit,E>}, {<intent, example>},{<exp,drawback>}, {<drawback,exp>,<exp,intent>}, {<benefit,exp>}

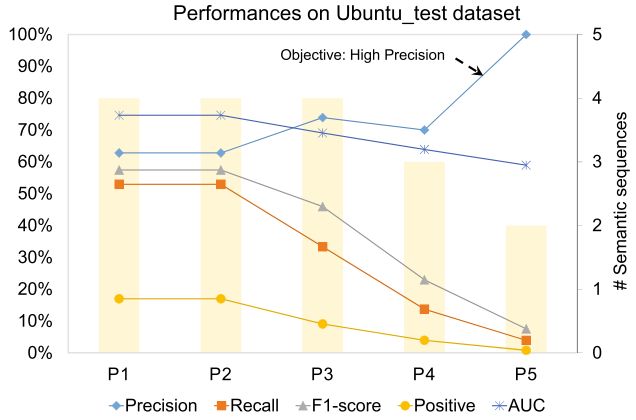

Fig. 2 Performance of 'high precision' patterns on non-fitted Ubuntu testing dataset

Summary The Although P5 can achieve 10 patterns mined from the Ubuntu train dataset can effectively classify new emails in the Ubuntu-test dataset that they can reach the average level of precision and recall at 74% and 64%, with an overall accuracy of 69%.

4.2 Answering RQ2 (generalizability)

Since we use emails from the same project to evaluate the performance in RQ1, there may exist generalization issues that the high level of precision and recall only exists in the Ubuntu project. To investigate whether the obtained semantic sequence patterns can also apply to other projects, we further extend the testing dataset by including four other open source projects as introduced in Table 3.

Figure 4 shows the performances on the four other open source projects, along with the performance on the Ubuntu test dataset. We highlight the patterns that can achieve relatively good performances. For high prediction patterns (P1–P5), we can see that performances on

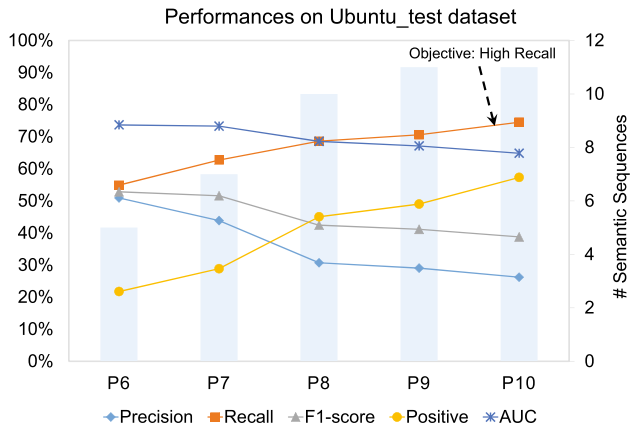

Fig. 3 Performance of 'high recall' patterns on non-fitted Ubuntu testing dataset

the four new open source projects are similar or even better than the Ubuntu-test dataset, where Jetty has better performance than the Ubuntu testing dataset among Precision, Positive, and AUC. Considering the five patterns, **P2 might be a better trade-off choice for high precision purpose**. Both AUC and F1-score are sharply declining after P2, but the values of precision except Jetty remain slightly changed after P2. For high recall patterns (P6–P10), we can see almost all the four new projects achieve better performances than the Ubuntu testing dataset. We consider that **P7 might be a better trade-off choice for high recall objective**. The values of AUC and F1-score on most projects largely declined after P7, but the values of recall increase slowly.

We also observe that the patterns achieve good performances on the Ubuntu test dataset, as well as the other four projects. Moreover, the patterns achieve even better results in terms of precision, recall, and F1-score on the other four projects. For example, the Jetty project achieves almost the best performances in all the metrics, which indicates that contributors are likely to use certain semantic sequences when expressing feature requests. This phenomenon confirms that the patterns mined from the Ubuntu community can also work well on other textual resources. We can also infer that, even though there are wide cultural diversities in a large open source community, contributors are likely to follow some common patterns when describing feature requests in emails.

Summary The patterns trained from the Ubuntu dataset are also suitable for other projects. The average of *precision* for high prediction patterns among the four new testing dataset is 77%, and the average of *recall* for high recall patterns is 91%, which confirms that contributors from the different community are likely to follow similar patterns when expressing feature request in emails.

4.3 Answering RQ3 (advantage)

In this section, we build prediction models with seven different existing classification approaches, and use these models to discover feature requests in the five testing dataset. We compare their performances with our extracted patterns. All the learning-based approaches are trained from the Ubuntu training dataset except DECA. DECA uses linguistic rules that are already built from other projects, which does not have any training process.

Figure 5 illustrates the max, min, and mean performances of the seven classification approaches on testing projects. For *precision*, we can see that the high-precision patterns (i.e., P1–P5) significantly outperform the other baselines. P3, P4, and P5 have the highest precision results, while most of the learning-based approaches and DECA are below 50%. For *recall*, we can see that the high-recall patterns could

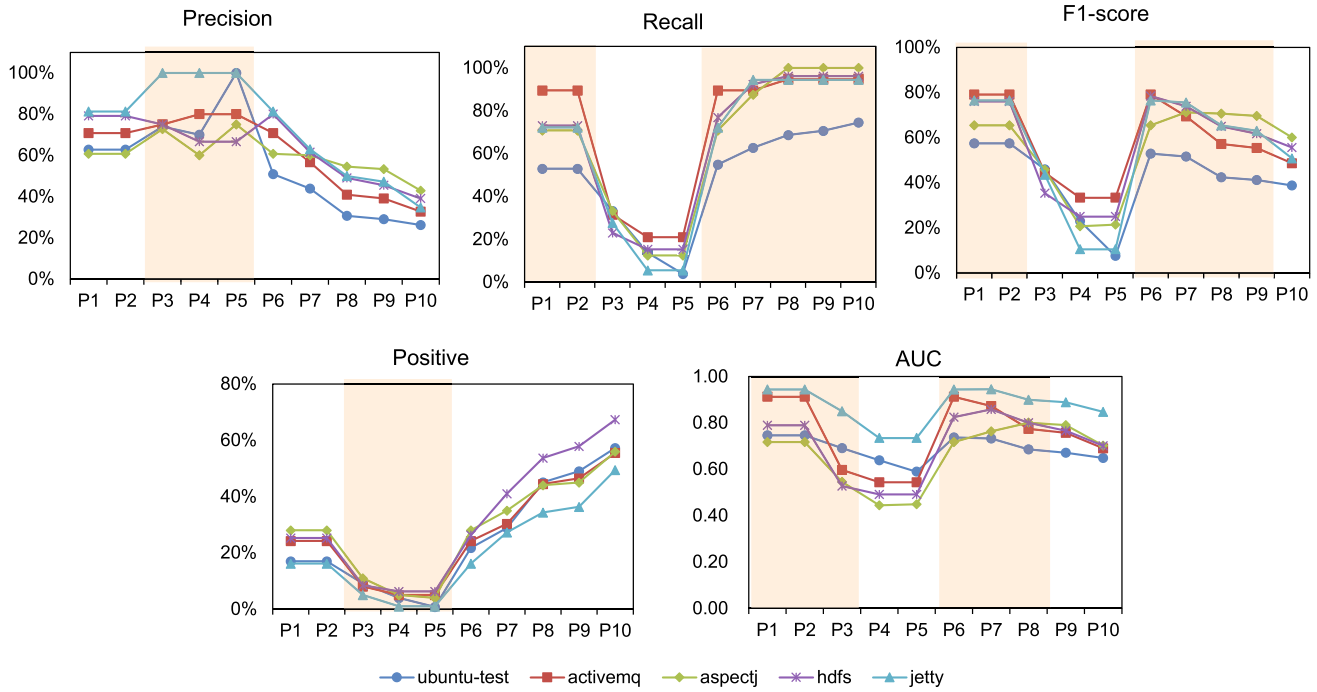

Fig. 4 Performance for patterns on cross-projects

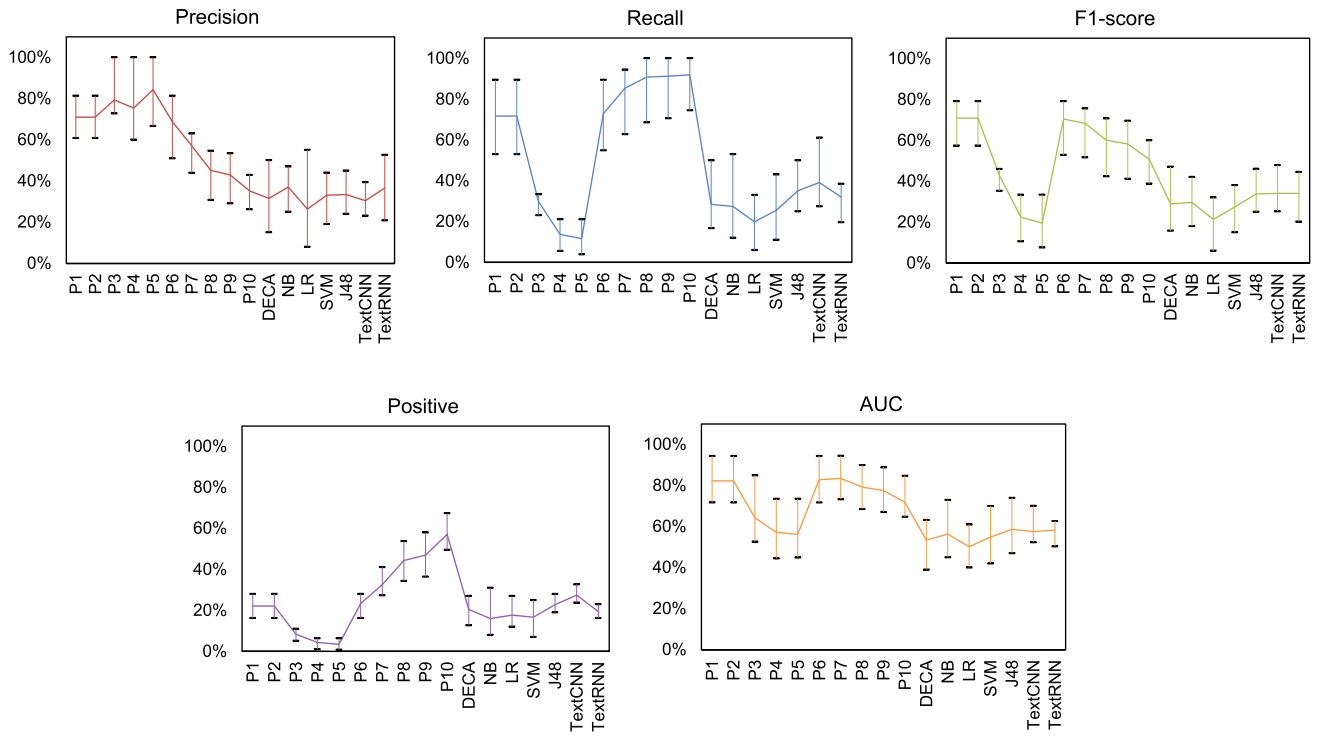

Fig. 5 Performance comparison between semantic sequence patterns and other approaches

achieve high recall values among most of the five projects. Meanwhile, P1 and P2 could also achieve relatively good results. DECA and six learning-based approaches are mostly below 60%. For *F1-score*, P1, P2, P6, P7, P8, and P9 are higher than other approaches. For *positive*, we can see that P3, P4, and P5 could reach the lowest values, while the precision is 100% correct as shown in the precision figure. P1 and P2 have similar positive values with DECA and the six learning-based approaches, but their F1-score and AUC are much higher. For *AUC*, we can see that the distribution of high performances is similar to the F1-score figure. Most of our patterns are above 0.7.

We further analyze why the DECA approach could not work well on the feature-request emails identification task. First, DECA utilized 36 linguistic rules [e.g., (someone) want to (something), and (something) should/could be (verb)] to identify sentences expressing feature requests. Those rules focus only on the lexical patterns of sentences, while the FRA tagging technique we adopted focuses on lexical, syntax, and semantic patterns. In some cases, pure lexical patterns could not provide precise annotations. For example, the sentence 'It makes harder for me when I want to 'cd' those directories in the terminal' is classified as a feature request by DECA, as it matches with the '[someone] wants to [something]' lexical pattern. But it is expressing complaints about the CD operation. FRA classifies this sentence with 'Drawback,' which provides a more accurate annotation result for further analysis. Second, our approach mines the semantic sequence patterns of a given email, which could obtain more rich information about the context of each sentence. By considering the context information, our approach has the opportunity to identify feature-request emails more accurately. For example, an email contains the sentence 'I need some advice with message groups and I've failed to find a solution in the net.' The sentence is tagged as feature-request by DECA, and the email containing the sentence is classified as a feature request as well according to our experiment definition. But the email is asking for advice rather than requesting features. When analyzing the contextual sentences, our approach could not match the email with our semantic sequence patterns. Thus, a negative prediction is made by our approach, which is a correct prediction in this case.

For the six learning-based approaches, the average F1-score is only around 30%. The reasons why our approach noticeably outperforms the six learning-based text classification models are: those text classification algorithms are not trained sufficiently from the limited training datasets, while our approach mine the expressing logic, i.e., the semantic sequence patterns, which is easier to train than the text classification tasks.

Summary When predicting whether emails are feature requests, most of our patterns outperform the existing approaches, and could achieve good performances in terms of precision, recall, F1-score, and AUC.

5 Discussion and future work

In this section, we discuss the implications of our results and possible ideas for future work.

5.1 Implications on how developers describe intentions

In our study, we tag each sentence in the feature-request emails into different categories according to its content. The sequence of tags forms a semantic sequence, which denotes the semantic logicality on how developers express feature requests. The semantic sequence patterns indicate the frequent logic flows that are commonly appearing in expressing feature requests. Unlike traditional ML approaches, the outputs of our approach are rule-based patterns, thus making them well suited for the software engineering tasks. As one of the results of our study, 10 semantic sequence patterns are identified by leveraging mining algorithms and objective optimization strategies. The result indicates that users are likely to follow some logical patterns to describe their intentions when requesting features. We believe that the proposed approach can contribute to the logic-based learning approaches [3].

By further analyzing the generated semantic sequence patterns, we might be able to classify feature requests according to their expression logics. For example, P5 provides two elements. One is {<S,intent>, <intent, E>}, and another is {<S, intent>, <drawback, expression>, <expression, intent>}. The first pattern may match emails that describe completely new ideas. The second pattern may match emails that complain about the existing system functionalities. Thus we may classify emails into new-idea feature requests and complaint feature requests based on the matching patterns. Moreover, analysis of the generated semantic sequence patterns may empirically confirm the existing personality and psychology studies on how people describe intentions. For example, five conditions have been reported on intentional behaviors determination [24, 34, 35]: (1) a desire for an outcome, (2) beliefs about a behavior leading to that outcome, (3) a resulting intention to perform that behavior, (4) the skill to perform the behavior, and (5) awareness of fulfilling the intention. In the generated patterns, we can recognize sequences that deliver similar information. For example, in P9, <S,intent> might fit condition

(1), <intent, benefit> might fit condition (2), and <intent, expression> might fit the other three conditions. Our results can provide practical evidence for those proposed models.

5.2 Balancing the cost and risk

Discovering feature requests from a large amount of emails is important for both open source and industrial organizations to maintain and improve their systems. Instead of analyzing every posted email, our patterns can narrow down the analysis scope to a smaller set. The 10 different patterns provided in our study can accommodate various objectives. For organizations that would like to obtain as many feature requests as they desire, P10 can be applied and gives a 92% recall on feature request emails by predicting 57% emails to be feature request emails (shown in Fig. 3). For organizations that would like to obtain a more accurate prediction on feature request emails, they can select P1–P5. Although P5 can achieve 84% precision on predicting feature request emails, it can only achieve the recall of 12%. P2 is more recommended since it can achieve 72% recall on the precision of 71%. Only 22% of the emails are predicted to be feature requests.

5.3 Extensive application

Social media, as the collective of online communication channels, includes many textual resources that may contain hidden feature requests. Our approach focuses on the textual content of development emails. It can also be extended to other similar resources, such as discussions in the issue tracking systems, comments in open forums, and project reviews. However, there are two limitations when applying our approach. First, our approach is constructed based on mining the semantic sequences extracted from textual artifacts. If the textual artifacts contain few semantic sequences, then the sequence mining algorithm will not work well and may be inefficient to find patterns. Therefore, the input data need to be filtered by a threshold on the length of sentences. Based on our experience, a threshold over 4 is recommended. Second, our approach is designed to analyze natural language artifacts. For textual resources that contain a large proportion of source code, filtration of source code from natural language text is needed before applying the proposed approach. The study proposed by Bacchelli et al. [6] on classifying email contents can help filter out non-natural language text emails. Moreover, the proposed approach can also help rebuild software requirements specifications for open source projects, which are often found lacking of high-quality documentation.

In future work, we plan to enlarge the tag categories by investigating more semantic categories. For example, we can apply sentiment analysis to define 'like' and 'hate' tags. Then we can use the enlarged tags to generate new patterns, and try to improve the prediction performances of the generated patterns. We also plan to conduct further analysis of the generated patterns to report how developers express feature requests empirically. In addition, we can refine feature requests into detail topics, such as new-idea feature requests and complaint feature requests by leveraging the matching patterns.

6 Threats to validity

External Validity The results of our study may not generalize beyond the project we evaluated. However, emails are found to be quite similar among different open source communities. To mitigate this threat, we had a set of criteria when selecting suitable projects. We picked Ubuntu, which is a popular and well-known open source project that has a large international community of contributors and a mature communication system via mailing lists. Moreover, we perform cross-project validation on the other four open source projects. The results show that our approach can also be generalizable to these projects, which largely reduces the external threat.

Internal Validity Threats to internal validity may come from the process of manually labeled emails to be feature requests or not. The accuracy of labeling has impact on our results. We understand that such a process is subject to mistakes. To reduce the threat, we build an inspection team to reach agreements on different options as introduced in Sect. 3.3.3. To answer RQ3, we rely on human judgments to classify emails, which is an error-prone process. To alleviate this threat, we gave a set of very specific instructions to each participant to clarify and unify the annotation process. Another threat may lie in the direct usage of existing work to automatically tag sentences in emails. Since the existing tagging approach is not one hundred percent correct, bias may be introduced by the misclassification of the tagging approach. However, the tagging approach has been proved to perform steadily well on completely new projects (with 84–87% correctness) by Shi et al. Moreover, we randomly sampled 100 auto-tagged sentences in emails, and 85% of them were correct. We consider that the tagging results are also reliable in feature request emails. Therefore, this threat has minor impact to the results.

Construct Validity The construct threats relate to the suitability of evaluation metrics. We utilize precision and recall to evaluate the performance, in which we use the manually labeled emails as ground-truth when calculating the performance metrics. The threats might come from the process of manual inspection and labeling. We understand that such a process is subject to mistakes. To reduce that

threat, we build two groups to reach agreements on different options.

7 Related work

7.1 Automated feature requests detection

Rodeghero et al. [36] presented an automatic technique that extracted useful information from the transcripts of developer-client spoken conversations to construct user stories. They used machine learning classifiers to determine whether a conversation contains user story information or not.

Maalej and Nabil [33] leveraged probabilistic techniques as well as text classification, natural language processing, and sentiment analysis techniques to classify app reviews into bug reports, feature requests, user experiences, and ratings. Their results showed that the classification can reach the precision between 70 and 95% and recall between 80 and 90% actual results.

Vlas and Robinson [55] proposed a grammar-based design of software automation for the discovery and classification of natural language requirements found in open source projects repositories. Herzig et al. [26] manually examined more than 7000 issue reports and discussed the impact of misclassification of bugs in the bug databases of five open source projects. Their results showed that 39% of files marked as defective actually include new features, updates to documentation, or internal refactoring. The authors suggested that humans should always be involved when dealing with the posted issue reports. Merten et al. [38] investigated natural language processing and machine learning features to detect software feature requests in issue tracking systems. Their results showed that software feature requests detection can be approached on the level of issues and data fields with satisfactory results. Merten et al. [37] also investigated how requirements are communicated in issue tracking systems by manually reviewing 200 issues. They categorized the text and reported on the distribution of issue types and information types. Their results showed that information with respect to prioritization and scheduling can be found in natural language data. Antoniol et al. [4] investigated whether the text of the issues posted in bug tracking systems is enough to classify them into corrective maintenance and other kinds of activities. They alternated among various machine learning approaches such as decision trees, naive Bayes classifiers, and logistic regression to distinguish enhancement apart from other issues posted in the system.

Cledland-Huang et al. [15] designed automatic forum management (AFM) system, which was used to automatically detect duplicated feature requests that have been already posted in the issue tracking systems. Lin et al. [48] proposed an approach to automatically identify redundant feature requests that have requested features that have been already implemented by applying the feature tree model.

Summing up, previous approaches differ from our work as they:

- identified feature requests from spoken conversations [36];
- identified feature requests from app reviews [33];
- identified feature requests from project repositories [55] and issue tracking systems [4, 26, 37, 38]
- detected duplicated and redundant feature requests in issue tracking systems [15, 48].

We retrieve hidden feature requests from development emails, which complements the existing studies on automatically detecting feature requests.

7.2 Email contents analysis

Di Sorbo et al. [52] proposed a classification approach to classify the sentences in development emails according to their purposes using natural language parsing techniques.

Furthermore, Huang et al. [28] found that Di Sorbo's work cannot be generalized to discussions in issue tracking systems, and they addressed the deficiencies of Di Sorbo et al.'s taxonomy by proposing a convolution neural network (CNN)-based approach. According to the purposes of sentences in emails, their work identified specific email fragments that can be used for specific maintenance tasks. Their work can classify Feature Requests at the sentence level, whereas we take the contextual information into consideration and focus on identifying the main intent of an email instead of email fragments.

Bacchelli et al. [6] presented an automatic approach to classify email content at line level into five language categories: Natural Language text, source code, stack traces, code patches, and junk. The results were validated through cross mailing list validation. Their work focuses on purifying the emails by finding natural language text, while we concern about whether an email contains any feature requests. Our study can complement their outputs as a deeper analyzer. Kiritchenko and Matwin [30] presented a paper on email classification by combining labeled and unlabeled data. Authors tried to define classes as interesting and uninteresting categories. VSM was showed to benefit from the co-training process proposed in the paper. They predict emails to be interesting or uninteresting, while we predict emails to be feature request or non feature request. Zhang et al. [57] extracted information from mailing lists to predict software defects. Metrics were summarized from the information. The results showed that defects were related to specific structures that appeared in mailing lists such as the content and thread structures.

They predict defects from emails, while we predict feature requests. Corston-Oliver et al. presented an approach to provide task oriented-summary of email messages by identifying task-related sentences in development messages. Mining of intention in developers' discussions provides a higher level of abstraction. The intents of the sentences are used to summarize tasks in their work, while we analyze the intents based on feature requests to identify semantic sequence patterns of feature requests.

Morales-Ramirez et al. [39] combined speech-act annotation and sentiment features to identify intentions, deontic mood, and length of the textual user feedbacks through sentence parsing. Multiple types of speech-act were used to indicate different purposes. For example, the combination of Requestive and Positive speech-acts may give a hint of a possible requirement. Although their work provided information regarding the correlation between speech-acts types and the issue types, it didn't provide a way to extract features from these issues and comments.

7.3 Automatic email classification

Emails are one of the crucial sources of communication, and the volume continues to grow. Many researches devote to automatic email classification, such as spam detection, multi-folder categorization, and phishing email classification.

Spam detection aims to develop binary classifiers that classify emails into spam or ham. Barushka and Hajek [10] proposed a regularized deep multi-layer perceptron neural network as a binary classifier, with 99.89% of accuracy on the SpamAssassin dataset and 98.76% of accuracy on the Enron-Spam dataset. Bahgat et al. [9] proposed a model based on a semantic feature selection with an SVM classifier, with 94% accuracy on Enron-Spam Dataset. Faris et al. [27] presented a binary model based on a Genetic Algorithm as a feature selector and the Random Weight Network as a classifier, which reached 96.70% of accuracy on the SpamAssassin dataset.

Multi-folder categorization approaches proposed multi-class classifiers that categorize emails into various user-defined email directories. Kiritchenko et al. [31] employs temporal features, such as day of the week and time of the day, to classify email messages into classes. They extracted relevant temporal features from emails and combined them with conventional content-based classification approaches. Chakravarthy et al. [14] presented a supervised learning approach that leverages graph mining techniques for multi-folder email classification. The experimental results showed significant performance improvement over Naive Bayesian approach for varied emails drawn from different domains. Aery et al. [1] proposed graph mining techniques for binary classification of documents in a delimited context, based on the occurrence of terms in the structured emails. The proposed approach outperformed the Naive Bayes and reached more than 90% of accuracy with a large size of folders.

Phishing email classification researches provided binary classifiers that categorize emails into phishing or ham. Fang et al. [19] proposed THEMIS which incorporated recurrent convolutional neural networks (RCNN) with multilevel vectors and attention mechanism. They modeled emails at the email header, the email body, the character level, and the word level simultaneously. The experimental results show that the overall accuracy of THEMIS reaches 99.848%. Bagui et al. [8] applied deep semantic analysis, machine learning, and deep learning techniques, to capture inherent characteristics of email text, and classify emails as phishing or non-phishing. Sankhwar et al. [45] proposed EMUD which focused on relevant URLs features that discriminate between legitimate and malicious/phishing URLs. This EMUD algorithm selects 14 heuristics to detect malicious or phishing URLs.

Our work differs from existing researches in that we focus on classifying emails into feature-request and non-feature-request, which aims to benefit the release planning practices. In addition, our work complements the existing studies on automatic email classification.

8 Conclusion

In this paper, we presented an approach to predict whether a development email is requesting features. After tagging each sentence in emails with the semantic categories based on 81 fuzzy rules extracted from confirmed feature requests by our previous work, we obtained the semantic sequences for each email. We then applied sequence pattern mining together with objective optimization to generate a set of 10 semantic sequence patterns.

To evaluate our approach, we conducted an empirical study on 248 emails from the Ubuntu community, which were not included in the training dataset. The results show that the 10 generated patterns can effectively identify feature requests from development emails while achieving different objectives. To achieve high precision, the corresponding patterns range the precision from 59 to 100% by predicting less than 18% emails to be feature requests. To achieve high recall, the corresponding patterns range the recall from 53 to 73% by keeping the AUC above 0.64.

We applied a state-of-the-art approach DECA and four machine learning approaches to classify the same 248 emails, and compared the performances with three of the 10 patterns we generated. The results show that compared with the five existing approaches, our approach outperformed in terms of precision, recall, F1-score, AUC, and positive.

We believe that our contributions can make the feature requests easier to understand and analyze, and can help to discover feature requests from massive textual emails, thus improve feature request management in the open source communities, as well as contribute to crowd-based requirements engineering.

Acknowledgements Our deepest gratitude goes to the anonymous reviewers for their careful work and thoughtful suggestions that have helped improve this manuscript substantially. We also would like to thank Michael Shoga for constructive criticism of this manuscript. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1403400, Youth Innovation Promotion Association CAS, and the National Science Foundation of China under Grant Nos. 61802374, 61432001, 61602450, and 62002348.

This material is also based upon work supported by the U.S. Department of Defense through the Systems Engineering Research Center (SERC), and the National Science Foundation Grant CMMI-1408909, Developing a Constructive Logic-Based Theory of Value-Based Systems Engineering.

References

1. Aery M, Chakravarthy S (2005) emailsift: email classification based on structure and content. In: Proceedings of the 5th IEEE international conference on data mining (ICDM 2005), 27–30 Nov 2005, Houston, Texas, USA, pp 18–25
2. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. Acm Sigmod Rec 22:207–216
3. Alrajeh D, Russo A, Uchitel S, Kramer J (2016) Logic-based learning in software engineering. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, pp 892–893
4. Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) 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, p 23
5. Antoniol G, Ayari K, Penta MD, Khomh F, Guéhéneuc Y (2008) Is it a bug or an enhancement? A text-based approach to classify change requests. In: Proceedings of the (2008) conference of the centre for advanced studies on collaborative research, Oct 27–30, 2008. Richmond Hill, p 23
6. Bacchelli A, Sasso TD, D'Ambros M, Lanza M (2012) Content classification of development emails. In: International conference on software engineering, pp 375–385
7. Bacchelli A, Mocci A, Cleve A, Lanza M (2017) Mining structured data in natural language artifacts with island parsing. Sci Comput Program 150:31–55
8. Bagui S, Nandi D, Bagui SC, White RJ (2019) Classifying phishing email using machine learning and deep learning. In: 2019 International conference on cyber security and protection of digital services, cyber security 2018, Oxford, United Kingdom, June 3–4, 2019, pp 1–2
9. Bahgat EM, Rady S, Gad W, Moawad IF (2018) Efficient email classification approach based on semantic methods. Ain Shams Eng J 9(4):3259–3269
10. Barushka A, Hajek P (2018) Spam filtering using integrated distribution-based balancing approach and regularized deep neural networks. Appl Intell 48(10):35383556
11. Bergstra J, Bengio Y (2012) Random search for hyper-parameter optimization. J Mach Learn Res 13(1):281–305
12. Brown PF, Desouza PV, Mercer RL, Pietra VJD, Lai JC (1992) Class-based n-gram models of natural language. Comput Linguist 18(4):467–479
13. Burdukiewicz M, Sobczyk P, Lauber C (2015) N-gram analysis of biological sequences. Biol Cybern 9(3):85–95
14. Chakravarthy S, Venkatachalam A, Telang A (2010) A graph-based approach for multi-folder email classification. In: ICDM 2010, the 10th IEEE international conference on data mining, Sydney, Australia, 14–17 Dec 2010, pp 78–87
15. Cleland-Huang J, Dumitru H, Duan C, Castro-Herrera C (2009) Automated support for managing feature requests in open forums. Commun ACM 52(10):68–74
16. Community U (2017) Mailing lists. https://lists.ubuntu.com/
17. Community U (2017) Ubuntu development discuss. https://lists.ubuntu.com/archives/ubuntu-devel-discuss/
18. Di Sorbo A, Panichella S, Visaggio CA, Di Penta M, Canfora G, Gall H (2016) Deca: development emails content analyzer. In: Proceedings of the 38th international conference on software engineering companion, ACM, ICSE '16, pp 641–644
19. Fang Y, Zhang C, Huang C, Liu L, Yang Y (2019) Phishing email detection using improved RCNN model with multilevel vectors and attention mechanism. IEEE Access 7:56329–56340
20. Fawcett T (2006) An introduction to ROC analysis. Pattern Recogn Lett 27(8):861–874
21. Goguen JA, Linde C (1993) Techniques for requirements elicitation. In: Proceedings of IEEE international symposium on requirements engineering, RE 1993, San Diego, California, USA, Jan 4–6, 1993, pp 152–164
22. Groen EC, Seyff N, Ali R, Dalpiaz F, Dörr J, Guzman E, Hosseini M, Marco J, Oriol M, Perini A, Stade MJC (2017) The crowd in requirements engineering: the landscape and challenges. IEEE Softw 34(2):44–52
23. Guzzi A, Bacchelli A, Lanza M, Pinzger M, Deursen AV (2013) Communication in open source software development mailing lists. In: Working conference on mining software repositories, pp 277–286
24. Heider F (1958) The psychology of interpersonal relations. Am Sociol Rev 23(6):170
25. Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: 35th International conference on software engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013, pp 392–401
26. Herzig K, Just S, Zeller A (2013) 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, pp 392–401
27. Faris H, Ala MAZ, Heidari AA, Aljarah I, Mafarja M, Hassonah MA, Fujita H (2018) An intelligent system for spam detection and identification of the most relevant features based on evolutionary random weight networks. Inf Fusion 48:67–83
28. Huang Q, Xia X, Lo D, Murphy GC (2020) Automating intention mining. IEEE Trans Softw Eng 46(10):1098–1119. https://doi.org/10.1109/TSE.2018.2876340
29. Kim Y (2014) Convolutional neural networks for sentence classification. In: Moschitti A, Pang B, Daelemans W (eds) Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, 2014, October 25-29, 2014, Doha, Qatar. Meeting of SIGDAT, a Special Interest Group of the ACL. ACL, pp 1746–1751. https://doi.org/10.3115/v1/d14-1181
30. Kiritchenko S, Matwin S (2011) Email classification with co-training. Ibm Corp 301–312
31. Kiritchenko S, Matwin S, Abu-Hakima S (2004) Email classification with temporal features. In: Intelligent information processing and web mining, proceedings of the international IIS: IIPWM'04

conference held in Zakopane, Poland, May 17–20, 2004, pp 523–533
32. Liu P, Qiu X, Huang X (2016) Recurrent neural network for text classification with multi-task learning. 1605.05101
33. Maalej W, Nabil H (2015) Bug report, feature request, or simply praise? On automatically classifying app reviews. In: 2015 IEEE 23rd international requirements engineering conference (RE), pp 116–125
34. Malle BF (1999) How people explain behavior: a new theoretical framework. Personal Soc Psychol Rev Off J Soc Person Soc Psychol 3(1):23
35. Malle BF, Knobe J (1997) The folk concept of intentionality. J Exp Soc Psychol 33(2):101–121
36. Mcmillan C, Mcmillan C, Mcmillan C, Mcmillan C (2017) Detecting user story information in developer-client conversations to generate extractive summaries. In: IEEE/ACM international conference on software engineering, pp 49–59
37. Merten T, Mager B, Hübner P, Quirchmayr T, Paech B, Bürsner S (2015) Requirements communication in issue tracking systems in four open-source projects. In: REFSQ workshops, pp 114–125
38. Merten T, Falis M, Hübner P, Quirchmayr T, Bürsner S, Paech B (2016) Software feature request detection in issue tracking systems. In: Requirements engineering conference (RE), 2016 IEEE 24th international, pp 166–175
39. Morales-Ramirez I, Kifetew FM, Perini A (2017) Analysis of online discussions in support of requirements discovery. In: International conference on advanced information systems engineering. Springer, Berlin, pp 159–174
40. Pei J, Han J, Mortazaviasl B, Pinto H, Chen Q, Dayal U, Hsu MC (2001) Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth, pp 215–224
41. Rijsbergen CJV (1979) Information retrieval, 2nd edn. Butterworth-Heinemann, Newton
42. Robertson AM, Willett P (1998) Applications of n-grams in textual information systems. J Doc 54(1):48–67
43. Russell SJ, Norvig PN (2010) Artificial intelligence: a modern approach. ThirdInternational Edition. Pearson Education. https://dblp.org/rec/books/daglib/0023820.bib
44. Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Inf Process Manag 24(5):513–523
45. Sankhwar S, Pandey D, Khan RA (2019) Email phishing: an enhanced classification model to detect malicious urls. EAI Endorsed Trans Scal Inf Syst 6(21):e5
46. Saraiva J, Bird C, Zimmermann T (2015) Products, developers, and milestones: How should i build my N-gram language model. In: Proceedings of the joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of Software Engineering (ESEC/FSE) Industry Track, ACM
47. Shi L, Wang Q, Li M (2013) Learning from evolution history to predict future requirement changes. In: 21st IEEE international requirements engineering conference, RE 2013, Rio de Janeiro, RJ, Brazil, July 15–19, 2013, pp 135–144
48. Shi L, Chen C, Wang Q, Boehm BW (2016) Is it a new feature or simply "don't know yet"?: On automated redundant OSS feature requests identification. In: 24th IEEE international requirements engineering conference, RE 2016, Beijing, China, Sep 12–16, 2016, pp 377–382
49. Shi L, Chen C, Wang Q, Li S, Boehm B (2017) Understanding feature requests by leveraging fuzzy method and linguistic analysis. In: IEEE/ACM international conference on automated software engineering, pp 440–450
50. Shi L, Chen C, Wang Q, Li S, Boehm BW (2017) Understanding feature requests by leveraging fuzzy method and linguistic analysis. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, Urbana, IL, USA, Oct 30–Nov 03, 2017, pp 440–450
51. Slimani T, Lazzez A (2013) Sequential mining: patterns and algorithms analysis. Int J Comput Electron Res 2(5):639–64
52. Sorbo AD, Panichella S, Visaggio CA, Penta MD, Canfora G, Gall HC (2015) Development emails content analyzer: intention mining in developer discussions (T). In: Proceedings of the 2015 30th IEEE/ACM international conference on automated software engineering (ASE), pp 12–23
53. Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. Springer, Berlin, pp 1–17
54. Steinmacher I, Silva MAG, Gerosa MA (2014) Barriers faced by newcomers to open source projects: a systematic review. In: Source Open Corral L, Sillitti A, Succi G, Vlasenko J, Wasserman AI (eds) Software, mobile open source technologies, pp 153–163
55. Vlas RE, Robinson WN (2012) Two rule-based natural language strategies for requirements discovery and classification in open source software development projects. J Manag Inf Syst 28(4):11–38
56. Zaki MJ (2001) Spade: an efficient algorithm for mining frequent sequences. Mach Learn 42(1–2):31–60
57. Zhang Y, Shen B, Chen Y (2014) Mining developer mailing list to predict software defects, vol. 1, pp 83–390