Development emails content analyzer: intention mining in developer discussions

Di Sorbo, Andrea ; Panichella, Sebastiano ; Visaggio, Corrado Aaron ; Di Penta, Massimiliano ; Canfora, Gerardo ; Gall, Harald

Abstract: Written development communication (e.g. mailing lists, issue trackers) constitutes a precious source of information to build recommenders for software engineers, for example aimed at suggesting experts, or at redocumenting existing source code. In this paper we propose a novel, semi-supervised approach named DECA (Development Emails Content Analyzer) that uses Natural Language Parsing to classify the content of development emails according to their purpose (e.g. feature request, opinion asking, problem discovery, solution proposal, information giving etc), identifying email elements that can be used for specific tasks. A study based on data from Qt and Ubuntu, highlights a high precision (90%) and recall (70%) of DECA in classifying email content, outperforming traditional machine learning strategies. Moreover, we successfully used DECA for re-documenting source code of Eclipse and Lucene, improving the recall, while keeping high precision, of a previous approach based on ad-hoc heuristics.

DOI: https://doi.org/10.1109/ASE.2015.12

Development Emails Content Analyzer: Intention Mining in Developer Discussions

Andrea Di Sorbo*, Sebastiano Panichella†, Corrado A. Visaggio*,
Massimiliano Di Penta*, Gerardo Canfora* and Harald C. Gall†
*University of Sannio, Benevento, Italy
†University of Zurich, Switzerland
disorbo@unisannio.it, panichella@ifi.uzh.ch, {visaggio,dipenta,canfora}@unisannio.it, gall@ifi.uzh.ch

***Abstract*—Written development communication (e.g. mailing lists, issue trackers) constitutes a precious source of information to build recommenders for software engineers, for example aimed at suggesting experts, or at redocumenting existing source code. In this paper we propose a novel, semi-supervised approach named DECA (Development Emails Content Analyzer) that uses Natural Language Parsing to classify the content of development emails according to their purpose (e.g. feature request, opinion asking, problem discovery, solution proposal, information giving etc), identifying email elements that can be used for specific tasks. A study based on data from Qt and Ubuntu, highlights a high precision (90%) and recall (70%) of DECA in classifying email content, outperforming traditional machine learning strategies. Moreover, we successfully used DECA for re-documenting source code of Eclipse and Lucene, improving the recall, while keeping high precision, of a previous approach based on ad-hoc heuristics.**

***Keywords*-Unstructured Data Mining, Natural Language Processing, Empirical Study**

I. Introduction

In many open sources and industrial projects, developers make an intense usage of written communication channels, such as mailing lists, issue trackers and chats [44]. Although voice communication still remains something unavoidable [1], [37], such channels ease the communication of developers spread around the world and working around the clock, and allows for keeping track of discussions and of decisions taken [8], [43]. From a completely different perspective, information contained in such a recorded communication has been exploited by researchers to build recommender systems, for example aimed at perform bug triaging [3], suggest mentors [11], or providing a description of an existing, undocumented software artifact [45].

However, profitably using information available in development communication is challenging, because of its noisiness and heterogeneity. Firstly, a development email or a post on issue tracker contains a mix of different kinds of structured, semi-structured, and unstructured information. For example, they may contain source code fragments, logs, stack traces, or natural language paragraphs mixed with some source code snippets, e.g. method signatures. In order to work effectively, recommenders must separate such elements, and this has been achieved by approaches combining machine learning techniques with island parsers [4], or by using other statistical techniques such as Hidden Markov Models [13].

The second issue is that communication posted on issue trackers, mailing lists or forums may have different purposes. For example, an issue report may relate to a feature request, a bug, or just to a project management discussion. For example, Herzig *et al.* [30] and Antoniol *et al.* [2] found that over 30% of all issue reports are misclassified (i.e., rather than referring to a code fix, they resulted in a new feature, an update of documentation, or an internal refactoring). Hence, relying on such data to build fault prediction or localization approaches might result in incorrect results. Kochhar *et al.* [35] shed light on the need for additional cleaning steps to be performed on issue reports for improving bug localization tasks. This, for example, may involve a re-classification of issue reports.

On a different side, certain recommender may require to mine specific portions of a written communication, for example to identify questions being asked by developers [29] or to mine descriptions about certain methods [5], [45]. Also, sometimes an email or a discussion is too long and this does not help a developer who get lost in unnecessary details. To cope with this issue, previous literature proposed approaches aimed at generating summaries of emails [36], [46], [48] and bug reports [47]. However, none of the aforementioned approaches is able to classify paragraphs contained in developers' communication according to the developers' intent, in order to only focus on paragraphs useful for a specific purposes (e.g. fixing bugs, add new features, improve existing features etc.).

Paper contribution. This paper proposes an approach, named DECA (*Development Email Content Analyzer*), that uses natural language parsing to capture linguistic patterns and classify emails' content according to developers' *intentions*, such as asking/providing helps, proposing a new feature or reporting/discussing a bug.

The use of natural language parsing is motivated by the need to better capture the intent of a sentence in a discussion, a task for which techniques based on lexicon analysis, such as Vector Space Models [6], Latent Semantic Indexing [18], or latent Dirichlet allocation (LDA) [10] would not be sufficient. For example, considering the following two sentences:

1) *We could use a leaky bucket algorithm to limit the bandwidth.*
2) *The leaky bucket algorithm fails in limiting the bandwidth.*

A topic analysis will reveal that these two sentences are likely to discuss the same topics: *"leaky bucket algorithm"* and *"bandwidth"*. However, these two sentences have completely different *intentions*: in sentence (1) the writer proposes a solution for a specific problem, while in sentence (2) the writer points out a problem. Thus, they could be useful in different contexts. This example highlights that understanding the *intentions* in developers' communication could add valuable information for guiding developers in detecting text content useful to accomplish different and specific maintenance and evolution tasks. For this reason, we devised six categories of sentences describing the *intent* of a developer (more details about the methodology used for the definition of the categories are in Section II-A): *feature request*, *opinion asking*, *problem discovery*, *solution proposal*, *information seeking* and *information giving*. Section II-B discusses how DECA detects, in accordance with these categories of sentences, developers' *intentions* behind the communication occurring during development, relying on Natural Language Parsing.

The main contributions of this paper are:

- A taxonomy of high-level categories of sentences, obtained by manually classifying development emails using grounded theory [25], along with a manually-labeled—and available for replication purposes—dataset of development emails from the Qt open source project and the Linux Ubuntu distribution.
- DECA, a novel automated approach to classify development emails content according to developers' intentions.
- A prototype implementation of the proposed approach available online.
- Results of an empirical comparison of DECA with machine learning classifiers.
- Last, but not least, as a practical application of DECA, we show how it can be used to mine method descriptions from developers' communication, and how DECA can overcome the limitations of a previously proposed approach [45].

The taxonomy proposed in this paper defines a conceptual framework for indexing discussions of different nature, as the inferred categories reflect some of the actual developers' needs in searching information across different channels. In this context, DECA could be very useful for classifying and indexing content of developers discussions occurring over several communication channels (i.e. issue trackers, IRC chats, on-line forums, etc.).

The results of our empirical study on data from Qt and Ubuntu, highlight a high precision (90%) and recall (70%) of DECA in classifying email content. Moreover, the proposed approach can be used for a wider application domain, such as the preprocessing phase of various summarization tasks. For example, DECA could be used as a preprocessing support to discard irrelevant sentences within emails or bug report summarization approaches [5], [36], [46], [48]. We successfully used DECA for re-documenting source code of Eclipse and Lucene, improving the recall, while keeping high precision, of a previous approach based on ad-hoc heuristics.

Paper structure. Section II presents the approach we defined to address the problem of email content classification. Section III reports the research questions of our empirical evaluation, the data we collected and the study design. Section IV discusses the results of this empirical study. Section V reports an evaluation of our approach in a real-life application. Section VI presents the threats that could affect the validity of our work. Section VII provides a discussion about the related literature and Section VIII concludes the paper outlining future research directions.

II. An Approach to Classify Emails According to Intentions

In this section we describe the approach we applied for the automatic classification of development emails content. In particular, we defined a taxonomy of sentences categories in order to catch useful contents for developers (see Section II-A). We extracted a set of linguistic patterns for each category. For each linguistic pattern we defined an heuristic responsible for the recognition of the specific pattern. We also developed a Java tool that implements the defined heuristics with the aim of enabling the automatic recognition of content fragments in development emails (see Section II-B).

A. Categories Definitions of Development Sentences

We have defined six categories of sentences describing the *"intent"* of the writer: *feature request, opinion asking, problem discovery, solution proposal, information seeking* and *information giving*. Table I provides a description of each category. These categories are designed to capture the aim of a given email and, consequently, recognizing the kind of information generally contained in messages regarding the development concerns.

TABLE I: Sentence Categories Definition

Category	Description
FEATURE REQUEST	Linguistic patterns related to ideas/suggestions/needs for improving or enhancing the product/service or its functionalities (e.g. "*We should add a new button*")
OPINION ASKING	Linguistic patterns for requiring someone to explicitly express his/her point of view about something (e.g. "*What do you think about the aspect of the main panel*")
PROBLEM DISCOVERY	Linguistic patterns related to issue definitions and unexpected behaviors (e.g. "*the problem occurs when I try to access to database*")
SOLUTION PROPOSAL	Linguistic patterns used to describe possible solutions for known problems (e.g. "*let's try to use a new method to compute costs*")
INFORMATION SEEKING	Linguistic patterns related to attempts to obtain information or help from other users (e.g. "*can you explain how the code works?*")
INFORMATION GIVING	Linguistic patterns exploited to inform/update other users about something (e.g. "*the plan is to release new updates in this week*")

The categories were identified by a manual inspection of a sample of 100 emails taken from the Qt Project development mailing list. During this task, we manually grouped all the extracted emails according to the categories defined by Guzzi *et al.* [26], which are: *implementation, technical infrastructure, project status, social interactions, usage* and *discarded*. We obtained 5 groups of emails, one for each category, with the

exception of *discarded*. The aim of the taxonomy presented in [26] was to assign topics to discussions' threads. Thus, this classification is useful to assign the scope to the entire mail message, but not the intent related to relevant sentences in the email content, which is our purpose. Indeed, we believe a single message may contain relevant sentences of different nature (e.g., an identification of a bug, and a subsequent solution proposal to fix it). Thus, for each group of emails we manually selected and extracted significant sentences that evoke, or suggest, the intent of the writer: e.g., is the writer saying that there is something to be implemented? Is the writer saying that she/he discovered a bug?

With the aim of defining a complete set of categories, we relied on a further taxonomy proposed by Guzzi *et al.* [27]. This second taxonomy classifies the reasons why developers need to communicate about source code. This taxonomy includes three categories: (i) *coordination*, (ii) *seeking information*, and (iii) *courtesy*. This second classification is close to the aim of our work, but it is not enough detailed. Through a manual inspection of the extracted sentences, we extended and refined the set of categories of Guzzi *et al.* [27] with a new set of categories (detailed in Table I), which better fits the content of development mailing list messages. We discarded the sentences that do not belong to any of the defined categories because they have negligible information (and/or are too generic) to help during a development task.

The categories we defined are intended to capture the intent of each sentence (requesting new features, describing a problem, or proposing a solution) and consequently allow developers to better manage the information contained in emails.

TABLE II: Sentence types by Guzzi et al. (left side) and ours (right side)

Example of Sentence Type	Guzzi's classification	Our categories
Discuss a change	Coordination	Feature Request
File a bug	Coordination	Problem Discovery
Know if a bug was fixed	Coordination	Information Seeking
Propose a solution for a known problem	Coordination	Solution Proposal
Ask other developers to make a change	Coordination	Feature Request
Ask how the code works	Seeking information	Information Seeking
Ask why the code was written in that way	Seeking information	Information Seeking
Ask an opinion to someone	Seeking information	Opinion Asking
Find out who wrote the code	Seeking information	Information Seeking
Learn more about the code	Seeking information	Information Seeking
Inform other developers about something	Courtesy	Information Giving

Table II compares our classification to the one proposed by Guzzi *et al.* [27], highlighting that our classification is more suitable for the software development domain, as it is more detailed and specific. As an example, let's consider three different kinds of sentences: (i) *discuss a change* (e.g., *"A computer icon is required"*, *"I would like to see a chat included in this release"*), (ii) *file a bug* (e.g., *"The server doesn't start"*, *"I found a problem during the installation process"*), and (iii) *propose a solution* (e.g., *"You may have to enable the package"*, *"What you need to do is to re-install the application"*). While in the classification of Guzzi *et al.* [27] all of them fall in the same category (*coordination*), in our model each of them is associated to a different category: *Feature Request, Problem Discovery*, and *Solution Proposal* respectively. We neglected some forms of *courtesy* (e.g., *ask a permission*) adding the *Information Giving* category to model cases in which the developers' intent is to provide useful information to other participants in the discussion (e.g., *"I have provided some extra information in the bug report"*, *"Plan is to make available a new release for this month"*). We finally introduced the *Opinion Asking* class for capturing explicit opinion requests (e.g., *"What do you think about creating a single webpage for all the services?"*). The sentences belonging to the *Opinion Asking* class may emphasize discussion elements which could be useful for developers' activities; thus, it appears reasonable to distinguish them from more general information requests (mapped with the *Information Seeking* category).

B. Natural Language Parsing of Linguistic Patterns

We assume that when developers write about existing bugs (*Problem Discovery*) or suggest solutions to solve these bugs (*Solution Proposal*) within discussions about development issues, they tend to use some recurrent linguistic patterns. For instance, let's consider again the example sentences introduced in Section I. Observing the syntax of the sentence *"We could use a leaky bucket algorithm to limit the bandwidth"*, we can notice that the sentence presents a well-defined *predicate-argument* structure:

- the verb "to use" constitutes the principal predicate of the sentence;
- "could" is the auxiliary of principal predicate.
- "we" represents the subject of the sentence;
- "a leaky bucket algorithm" represents the direct object of the principal predicate;
- "to limit" is a non-finite clausal complement depending on the principal predicate;
- "the bandwidth" is the direct object of the clausal complement;

By exploiting this information, most of the sentences that present similar predicate-argument structure would indicate a *Solution Proposal*. Thus, we define a heuristic to detect this particular predicate-argument structure. The formalization of a heuristic requires three steps: (i) discovering the relevant details that make the particular syntactic structure of the sentence recognizable (e.g. the verb "to use" as principal predicate of the sentence and the auxiliary "could"), (ii) generalizing some kinds of information (e.g. subject doesn't necessarily be "we" and direct object doesn't necessarily be "leaky bucket algorithm"), (iii) ignoring useless information (e.g. the clausal complement and its direct object don't provide any useful information for the structure identification). So, we define a general pattern *"[someone] could use [something]"* (the words

in square brackets are placeholders indicating generic subjects, [someone], and generic direct objects, [something]) and associate it with the *Solution Proposal* class. On the contrary, if we consider the sentence *"The leaky bucket algorithm fails in limiting the bandwidth"*, we can notice that this second sentence has a totally different structure. Indeed, the verb "to fail" constitutes the principal predicate of the sentence and this would rather suggest the description of problem.

We used the Stanford typed dependencies (SD) representation [21] in order to describe a set of heuristics able to recognize similar recurrent linguistic patterns used by developers in an automated way. Each category is then associated to a group of heuristics. Thus, each heuristic is leveraged for the recognition of a specific linguistic pattern. The typed dependency parser represents dependencies between individual words contained in sentences and labels each of them with specific grammatical relation (such as subject or indirect objects) [19], [21]. The SD representation was successfully used in a range of tasks, including Textual Entailments [17] and BioNLP [24], and in recent years SD structures have also become a de-facto standard for parser evaluation in English [20], [12], [41].

Figure 1, through an example, shows the process we applied to define each heuristic. More precisely, in Figure 1 we can see how previously discussed concepts on sentences' predicate-argument structures can be implemented through the Stanford typed dependencies. Firstly, we analyze a sentence containing one of the recurrent linguistic patterns (e.g. *"We should add a new button to access to personal contents"* in Figure 1) and build its SD representation.

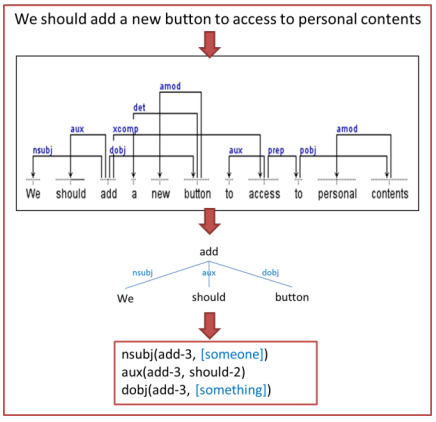


Fig. 1: Natural language parse tree from a Feature Request

TABLE III: Defined heuristics for each Sentence Category

Sentence Class	Heuristics
Feature Request	36
Opinion Asking	5
Problem Discovery	29
Solution Proposal	51
Information Seeking	57
Information Giving	53
TOTAL	**231**

In Figure 1, the main predicate, *"add"*, jointly to the auxiliary verb *"should"* is a clue for a *Feature Request*. Obviously this predicate has to be connected to a generic subject (that indicates who makes the request) and one (or more) generic direct object (that along with the predicate indicates the request object). At this point, we can define the related heuristic *"[someone] should add [something]"* and associate it with the *Feature Request* class. Once we have defined the heuristic, a sentence having a similar structure ("add" or synonyms in the role of principal predicate, "should" or synonyms in the role of auxiliary verb and one or more direct objects that indicates the things a user would add) can be recognized as belonging to the class of *Feature Request*. Through the process described above, we defined a set of heuristics[1] for each of the defined categories. Table III shows the number of heuristics implemented for each class.

We implemented a Java tool for evaluating the recognition capability of the proposed approach. By exploiting the defined heuristics and working on the SD representation of the emails' sentences, the tool highlights the recognized sentences with different colors. We make the tool[2] and a replication package[3] available to help other researchers to easily replicate our study.

III. Evaluation: Study Definition

The *goal* of this study is to analyze development emails contents with the *purpose* of investigating the effectiveness of the approach in identifying discussions relevant for developers for specific maintenance task.The *perspective* is of researchers interested in identifying relevant recurring linguistic patterns in the software engineering domain, useful for performing several software engineering tasks.

A. Research Questions

The study aims at investigating the following research questions:

- **RQ1**: *Is the proposed approach effective in classifying writers' intentions in development emails?*
  This research question represents the core part of our study aimed at developing and evaluating a novel approach for classifying messages able to help retrieving meaningful information from message content.
- **RQ2**: *Is the proposed approach more effective than existing ML in classifying development emails content?*
  This research question aims at comparing results achieved by DECA with the results obtained by a set of existing ML techniques previously used in the literature for classifying bug reports. Thus, this research question is aimed at quantifying the benefits obtained by the use of Natural Language Parsing with respect to existing ML techniques.

[1] http://www.ifi.uzh.ch/seal/people/panichella/DECA_Implemented_Heuristics.pdf
[2] http://www.ifi.uzh.ch/seal/people/panichella/tools/DECA.html
[3] http://cms.uzh.ch/lenya/ifi/authoring/seal/people/panichella/ReplicationpackageDECA.zip

B. Context Selection and Data Extraction

The context of the study consists of mailing lists belonging to two open source projects whose characteristics are summarized in Table IV. Specifically, for each project Table IV provides: (i) name, (ii) home page, (iii) period of time considered to collect the emails and (iv) total number of analyzed emails.

The Qt project is a cross-platform application and UI framework used to develop application software that can run on various software and hardware platforms. The development of Qt framework started in 1991 while Nokia founded the Qt Project on 21 October 2011 with the aim of easing online communication among Qt developers and community members through public forums, mailing lists and wikis.

Ubuntu is a Debian-based Linux operating system.

TABLE IV: Analyzed Projects

Project	Url	Observation Period	Emails analyzed
Qt Project	http://qt-project.org	March 2014 – September 2014	302
Ubuntu	http://www.ubuntu.com	September 2004 – January 2005	100

TABLE V: Development mailing lists samples

Project	Number of Messages	Posting Period
Qt Project	102	June 2014
Qt Project	100	May 2014
Qt Project	20	March 2014
Qt Project	20	April 2014
Qt Project	20	July 2014
Qt Project	20	August 2014
Qt Project	20	September 2014
Ubuntu	20	September 2004
Ubuntu	20	October 2004
Ubuntu	20	November 2004
Ubuntu	20	December 2004
Ubuntu	20	January 2005

Both projects have large development communities and this ensures high messages density (more than 100 messages per month). In order to have as many message types as possible in our study, we selected emails in specific time windows for the two projects. For the Qt Project we selected emails in a period related to a very advanced development stage (the development of Qt framework started in 1991) in which we expected to find more messages related to information requests and solution proposals. For Ubuntu we selected emails in a period related to a very early development stage (the first release of Ubuntu was issued in October 2004) in which we expected to find more messages related to new bugs discovered and/or feature requests.

Table V summarizes the samples of emails we randomly selected for our study, reporting for each sample the (i) name of the project, the (ii) amount of messages considered in the sample for that project and the (iii) period of time in which the messages of the sample have been exchanged. In this dataset, we anonymized the messages and applied a pruning of email metadata (removing for example, names of sender and receiver) that were not relevant for our goals.

Starting from sentences categories defined in Section II-A, two phD students (one of them not involved in this research work) separately analyzed all the messages in the dataset and manually extracted significant sentences assigning each of them to one of the defined categories. We involved an external evaluator to avoid any bias related to the subjectivity of the classification. Specifically, the classifications performed by the two evaluators coincide for the most part (in about 97% of the cases they assigned a sentence to the same category). We considered only the sentences that both evaluators assigned to the same category. Table VI shows the samples' size of the classified sentences for the two projects considered in our study. It is important to stress that the proportion of

TABLE VI: Samples Size of Classified Sentences

Sentence class	Qt Project	Ubuntu
Feature request	151 (18.50%)	54 (20.69%)
Opinion asking	9 (1.10%)	8 (3.07%)
Problem discovery	132 (16.18%)	49 (18.77%)
Solution proposal	169 (20.71%)	34 (13.03%)
Information seeking	224 (27.45%)	66 (25.29%)
Information giving	131 (16.05%)	50 (19.16%)
Total	816	261

the categories of sentences varies depending on the projects. However, in both projects *Opinion Asking* and *Information Seeking* are respectively the categories having the lowest and the highest percentages of occurrences if compared to each other class of sentences.

C. Analysis Method

To answer RQ1 we defined a sequence of train and test sets pairs for a progressive assessment of the results. Thus, we scheduled 3 experiments.

Experiment I

a. We randomly selected as training set 102 emails among the messages sent in June 2014 related to Qt Project development mailing list. As explained in Section III-B, two humans performed the manual classification of the sentences contained in such email messages according to the defined category in Section II-A. Thus, we manually detected the recurring linguistic patterns found in this set of messages according to the defined categories in Section II-A. Through the process explained in Section II-B (Figure 1), we defined and implemented 87 heuristics for automatically classifying (recognition of patterns) the sentences contained in training set.

b. We randomly selected as test set 100 emails sent in May 2014 regarding the Qt Project. Also in this case, two people performed a manual classification of the contents of these messages according to the defined category. Specifically, only senteces evaluated as belonging to the same category were selected.

c. Relying on the 87 defined heuristics, we used our tool to automatically classify sentences of the emails in the test set. We compared tool outputs with the human generated oracle

and computed (i) *true positives* (*TP*) as the number of sentences correctly classified by the tool, (ii) *false positives* (*FP*) as the number of sentences incorrectly labeled as belonging to a given class, and (iii) *false negatives* (*FN*) as the number of items, which were not assigned to any sentence category but were belonging to one of them. Thus, we evaluated the tool performances relying on the widely adopted metrics of Information Retrieval: Precision, Recall and F-Measure.

Experiment II

a. To improve the effectiveness of the DECA's classification we used the set of sentences classified as false negatives in the Experiment I as a gold set for defining new heuristics able to capture such sentences. Specifically, 82 new heuristics were identified, formalized and implemented in order to detect the sentences not identified in Experiment I. Thus, our heuristics set increased from 87 to 169.
b. We prepared a new test set to verify if the augmented heuristics set allowed us to get better results. 100 emails were randomly selected between messages sent in the months of March, April, July, August, September of the year 2014 related to Qt Project. Following the same approach previously discussed, two human judges contributed to constitute the oracle for this experiment.
c. We executed DECA on emails sentences of this new test set and we compared the data with the human generated oracle.

Experiment III

a. To further improve the effectiveness of the classification performed by DECA, we used again false negatives found in the Experiment II as new set for identifying new recurrent patterns. In this way, 62 new heuristics were implemented, giving a total number of 231.
b. To evaluate the potential usefulness of the new set of heuristics, we created a third test set randomly selecting 100 emails sent from September 2004 to January 2005 in the Ubuntu distribution. Two human judges created the oracle table relatively to this test set, according to the previously explained process.
c. We executed DECA on the emails messages of this new test set and compared the results with the human generated oracle.

It is worth nothing that the two evaluators are the same in all the three experiments.

D. An Approach Based on ML for Email Content Classification

This section discusses the methodology we used to train machine learning techniques to classify the email content (RQ2). Specifically, the work by Antoniol *et al.* [2] exploited conventional text mining and machine-learning techniques to automatically classify bug reports. They used terms contained in bug reports as features (fields) of machine learning models to discern bugs from other issues. The work by Zhou *et al.* [58] extended the work of Antoniol *et al.* building the ML techniques considering, as additional features, structural information improving ML prediction accuracy. We implemented an approach, similar to the one used by Antoniol *et al.*, to classify sentences contained in mailing lists data using as features the terms contained in the sentences themselves.

Formally, given a training set of mailing list sentences $\mathbf{T}_1$ and a test set of mailing list sentences $\mathbf{T}_2$, we automatically classify the email content contained in $\mathbf{T}_2$, by performing the following steps:

1. *Text Features:* the first step uses all sentences contained in $\mathbf{T}_1$ and $\mathbf{T}_2$ as base information to build a textual corpus (indexing the text). In particular, we preprocessed the textual content applying stop-word removal and stemming [23] (similarly to the work of Zhou *et al.* [58]) to reduce the number of features for the ML techniques. The output of this phase is a Term-by-Documents matrix $\mathbf{M}$ where each column represents a sentence and each row represents a term contained in the given sentence. Thus, each entry $\mathbf{M}_{[i,j]}$ of the matrix represents the weight (or importance) of the $i-$th term contained in the $j-$th sentence. Similarly to the work of Antoniol *et al.* [2] we weighted words using the the tf (term frequency), which weights each words *i* in a document *j* as:

$$tf_{i,j} = \frac{rf_{i,j}}{\sum_{k=1}^{m} rf_{k,j}}$$

   where $\mathbf{rf}_{i,j}$ is the raw frequency (number of occurrences) of word *i* in document j. We used the *tf* (term frequency) instead of *tf-idf* indexing, because the use of the inverse document frequency (idf) penalizes too much terms appearing in too many documents [23]. In our work, we are not interested in penalizing such terms (e.g., "fix", "problem", or "feature") that actually appear in many documents because they may constitute interesting features that guide ML techniques in classifying development sentences.
2. *Split training and test features:* the second step splits the matrix $\mathbf{M}$ (the output of the previous step) in two submatrices $\mathbf{M}_{training}$ and $\mathbf{M}_{test}$. Specifically, $\mathbf{M}_{training}$ and $\mathbf{M}_{test}$ represent the matrix that contains the sentences (i.e., the corresponding columns in $\mathbf{M}$) of $\mathbf{T}_1$ and the matrix that contains the sentences (i.e., the corresponding columns in $\mathbf{M}$) $\mathbf{T}_2$ respectively.
3. *Oracle building:* this step aims at building the oracle to allow ML techniques to learn from $\mathbf{M}_{training}$ and predict on $\mathbf{M}_{test}$. Thus, in this stage, we manually classified the sentences in $\mathbf{T}_1$ and $\mathbf{T}_2$ assigning each of them to one of the categories defined in Section II-A (as described in Section III-C two human evaluators performed this classification). We added the value of the classification as further columns in both $\mathbf{M}_{training}$ and $\mathbf{M}_{test}$. The machine learning techniques during the training phase use the column **"C"** of the classification for learning the model.
4. *Classification:* this step aims at automatically classifying sentences relying on the output data obtained from the previous step ($\mathbf{M}_{training}$ and $\mathbf{M}_{test}$ with classified sentences). The automatic classification of sentences is

performed using the Weka tool [53] experimenting with eight different machine learning techniques, namely, the standard probabilistic naive Bayes classifier, the Logistic Regression, Simple Logistic, J48, the alternating decision tree (ADTree), Random Forest, FT, Ninge. The choice of these techniques is not random. We selected them since they were successfully used for bug reports classification [2], [58] (i.e., ADTree, Logistic regression) and for defect prediction in many previous works [7], [9], [14], [38], [59], thus allowing to increase the generalisability of our findings.

In the replication package[3], we make available the data we used for training and test the ML techniques. It is important to specify that the generic training set $\mathbf{T}_1$ and the generic test set $\mathbf{T}_2$, correspond to training and test set pairs discussed in Section III-C.

IV. Analysis of Results

A. RQ1:Is the proposed approach effective in classifying writers' intentions in development emails?

TABLE VII: Results for Experiment I

CLASS	TP	FN	FP	PRECISION	RECALL	F-MEASURE
FEATURE REQUEST	30	43	0	1.000	0.411	0.583
OPINION ASKING	0	0	0	0.000	0.000	0.000
PROBLEM DISCOVERY	37	37	9	0.804	0.500	0.617
SOLUTION PROPOSAL	19	52	5	0.792	0.268	0.400
INFORMATION SEEKING	29	73	2	0.935	0.284	0.436
INFORMATION GIVING	13	43	3	0.813	0.232	0.361
TOTAL	128	248	19	0.871	0.340	0.489

TABLE VIII: Results for Experiment II

CLASS	TP	FN	FP	PRECISION	RECALL	F-MEASURE
FEATURE REQUEST	32	26	7	0.821	0.552	0.660
OPINION ASKING	3	2	0	1.000	0.600	0.750
PROBLEM DISCOVERY	32	14	4	0.889	0.696	0.780
SOLUTION PROPOSAL	33	44	11	0.750	0.429	0.545
INFORMATION SEEKING	56	42	0	1.000	0.571	0.727
INFORMATION GIVING	41	20	4	0.911	0.672	0.774
TOTAL	197	148	26	0.883	0.571	0.694

TABLE IX: Results for Experiment III

CLASS	TP	FN	FP	PRECISION	RECALL	F-MEASURE
FEATURE REQUEST	46	8	9	0.836	0.852	0.844
OPINION ASKING	6	2	0	1.000	0.750	0.857
PROBLEM DISCOVERY	39	10	2	0.951	0.796	0.867
SOLUTION PROPOSAL	17	17	6	0.739	0.500	0.596
INFORMATION SEEKING	49	17	1	0.980	0.742	0.845
INFORMATION GIVING	26	24	2	0.929	0.520	0.667
TOTAL	183	78	20	0.901	0.701	0.789

Tables VII, VIII and IX report the results achieved by DECA in classifying development emails content. In particular, these tables show the amounts of (i) true positives, (ii) false negatives, (iii) false positives, (iv) precision, (v) recall and (vi) F-Measure achieved for each defined class for the three experiments (respectively). In general, the results of the classification performed by DECA are rather positive and the addition of new heuristics improves the effectiveness of the approach along the various experiments. Specifically, while the precision is always very high (ranges between 87% and 90%) and stable for all the experiments, the recall increases with the addition of new heuristics from 34% to 70% (i.e., around two times). This is also reflected by the increment of the F-Measure in the three experiments; it varies from an initial value of 49% (Experiment I) to 79% (Experiment III).

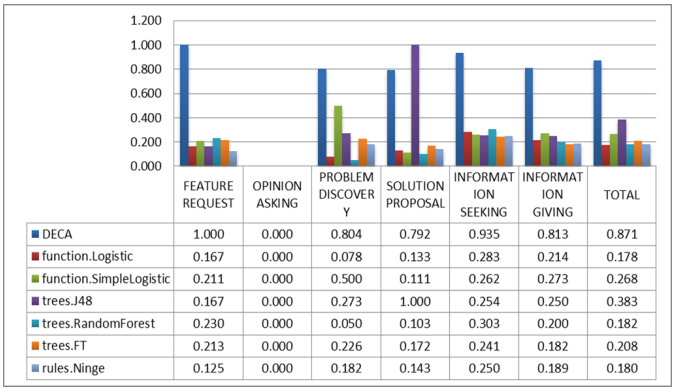
Furthermore, data shows that DECA works well for all the categories of sentences and all the experiments. The only exception is in the Experiment I for the Opinion Asking category, where recall and precision are equal to zero. However, in general for the Experiment I, precision ranges from 79.2% obtained for *Solution Proposal* to 100% achieved in *Feature Request*, whereas recall ranges from 23.2% for *Information Seeking* category to 50% achieved for *Problem Discovery*. In the Experiment II, precision ranges from 75% obtained for *Solution Proposal* to 100% achieved in *Information Seeking* and *Opinion Asking* categories, whereas recall ranges from 42.9% obtained for *Solution Proposal* to 69.6% achieved in *Problem Discovery*. In the Experiment III, precision ranges from 73.9% obtained for *Solution Proposal* to 100% achieved in *Opinion Asking* category, whereas recall ranges from 50% obtained for *Solution Proposals* to 85.2% achieved in *Feature Request*.

It is important to note that we achieved the best results in terms of recall classifying *Problem Discoveries* in the Experiments I and II. This indicates that developers very often rely on common/recurrent patterns successfully detected by DECA when their intent is to communicate a bug or a problem. On the other hand, we achieved the worst results in detecting *Solution Proposals* for all the three experiments with a precision that has gradually (and relatively) deteriorated (from 79.2% in Experiment I to 73.9% in Experiment III) and a recall that has gradually increased but never exceeded the 50%. This suggests that there are many different ways developers use when proposing solutions, making it hard to identify common patterns to detect them.

> **Summary RQ1**: *the automatic classification performed by DECA achieves very good results in terms of both precision, recall and F-measure (over all the experiments). The results tend to improve when adding new heuristics. DECA achieved the best values of F-Measure for Problem Discovery sentences and the worst F-Measure results for Solution Proposal sentences.*

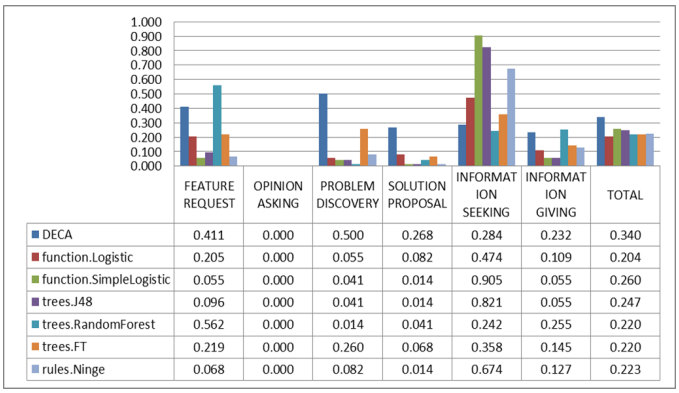
B. RQ2: Is the proposed approach more effective than existing ML in classifying development emails content?

As discussed in Section III-D, this research question aims at comparing performances of DECA with the performances of a set of machine learning techniques. For the lack of space, we report in this paper only the results of the ML models that obtained the best performances in classifying development content. Specifically, in order to get a more complete picture, we selected a set of techniques belonging to different ML categories: regression functions (i.e. Logistic Regression, Simple Logistic), decision trees (i.e. J48, FT, Random Forest), and rules models (i.e. Ninge). The comparison of results for the Experiment I highlights that DECA achieves the best global results in terms of both precision (see Figure 2) and



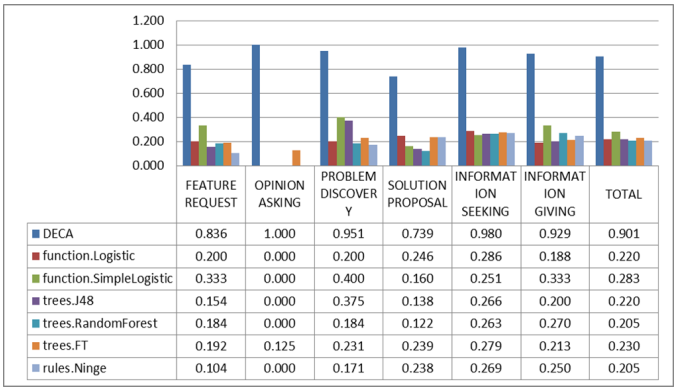
	FEATURE REQUEST	OPINION ASKING	PROBLEM DISCOVERY	SOLUTION PROPOSAL	INFORMATION SEEKING	INFORMATION GIVING	TOTAL
DECA	1.000	0.000	0.804	0.792	0.935	0.813	0.871
function.Logistic	0.167	0.000	0.078	0.133	0.283	0.214	0.178
function.SimpleLogistic	0.211	0.000	0.500	0.111	0.262	0.273	0.268
trees.J48	0.167	0.000	0.273	1.000	0.254	0.250	0.383
trees.RandomForest	0.230	0.000	0.050	0.103	0.303	0.200	0.182
trees.FT	0.213	0.000	0.226	0.172	0.241	0.182	0.208
rules.Ninge	0.125	0.000	0.182	0.143	0.250	0.189	0.180

Fig. 2: Compared Precision for Experiment I



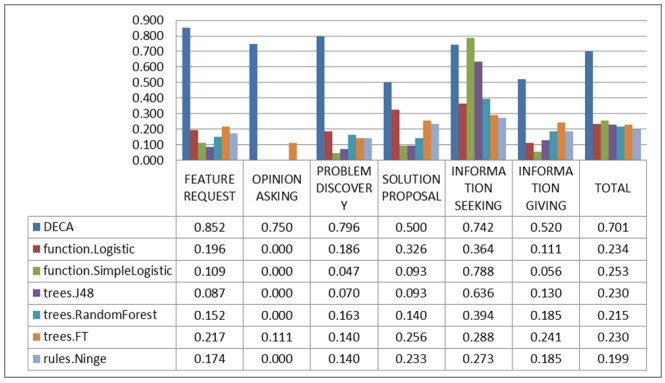
	FEATURE REQUEST	OPINION ASKING	PROBLEM DISCOVERY	SOLUTION PROPOSAL	INFORMATION SEEKING	INFORMATION GIVING	TOTAL
DECA	0.411	0.000	0.500	0.268	0.284	0.232	0.340
function.Logistic	0.205	0.000	0.055	0.082	0.474	0.109	0.204
function.SimpleLogistic	0.055	0.000	0.041	0.014	0.905	0.055	0.260
trees.J48	0.096	0.000	0.041	0.014	0.821	0.055	0.247
trees.RandomForest	0.562	0.000	0.014	0.041	0.242	0.255	0.220
trees.FT	0.219	0.000	0.260	0.068	0.358	0.145	0.220
rules.Ninge	0.068	0.000	0.082	0.014	0.674	0.127	0.223

Fig. 3: Compared Recall for Experiment I



	FEATURE REQUEST	OPINION ASKING	PROBLEM DISCOVERY	SOLUTION PROPOSAL	INFORMATION SEEKING	INFORMATION GIVING	TOTAL
DECA	0.836	1.000	0.951	0.739	0.980	0.929	0.901
function.Logistic	0.200	0.000	0.200	0.246	0.286	0.188	0.220
function.SimpleLogistic	0.333	0.000	0.400	0.160	0.251	0.333	0.283
trees.J48	0.154	0.000	0.375	0.138	0.266	0.200	0.220
trees.RandomForest	0.184	0.000	0.184	0.122	0.263	0.270	0.205
trees.FT	0.192	0.125	0.231	0.239	0.279	0.213	0.230
rules.Ninge	0.104	0.000	0.171	0.238	0.269	0.250	0.205

Fig. 4: Compared Precision for Experiment III



	FEATURE REQUEST	OPINION ASKING	PROBLEM DISCOVERY	SOLUTION PROPOSAL	INFORMATION SEEKING	INFORMATION GIVING	TOTAL
DECA	0.852	0.750	0.796	0.500	0.742	0.520	0.701
function.Logistic	0.196	0.000	0.186	0.326	0.364	0.111	0.234
function.SimpleLogistic	0.109	0.000	0.047	0.093	0.788	0.056	0.253
trees.J48	0.087	0.000	0.070	0.093	0.636	0.130	0.230
trees.RandomForest	0.152	0.000	0.163	0.140	0.394	0.185	0.215
trees.FT	0.217	0.111	0.140	0.256	0.288	0.241	0.230
rules.Ninge	0.174	0.000	0.140	0.233	0.273	0.185	0.199

Fig. 5: Compared Recall for Experiment III

recall (see Figure 3). The precision of DECA was only worse than J48 technique when identifying *Solution Proposal* (79% for DECA with respect to 100% for J48). However, for the *Solution Proposal* category DECA achieved a recall of 26.8% while J48 reached a recall of only 1.4% (see Figure 3). Focusing the attention on recall, our approach was worse than the RandomForest technique in detecting *Feature Request* (41.1% versus 56,2%, see Figure 3); on the other side, for *Feature Request* class DECA achieved a precision of 100% while RandomForest obtained a precision of only 23% (see Figure 2). Furthermore, the recall of our approach was better only than RandomForest technique in identifying *Information Seeking* sentences.

However, DECA results are much better than all the other techniques in terms of precision (around 94%, as showed in Figure 2). Finally, in the Experiment I, the recall achieved by DECA in detecting *Information Giving* sentences was comparable to the RandomForest technique (23.2% versus 25.5%). Also in this case DECA achieved a better precision (81.3% versus 20% of RandomForest).

In both the Experiment II and the Experiment III DECA achieved the best global results in terms of both Precision and Recall. We discuss in details the results of the comparison in Experiment III (Figure 4 and Figure 5).

Specifically, in the Experiment III DECA outperforms, in terms of recall, precision and F-measure the results of all the ML techniques. What is interesting to highlight is that our approach was the only technique able to recognize *Opinion Asking* sentences in the Experiment III (the same happened in Experiment II) with a substantially high precision and recall (precision 100% and a recall of 75%).

DECA obtained the best F-Measure values for all the defined sentences' categories in all the three experiments. To evaluate the performances of the proposed approach we repeated the run of each experiments 100 times. The results of the ML classifiers and DECA were pretty stable and statistically equal to the results showed in the study. In Experiment I DECA achieved an average F-Measure of 31% that is better than the F-Measure that can be achieved relying on all other considered techniques. While, in Experiment III DECA obtained an average F-Measure of 58% (49% for Experiment II) also in this case higher than the F-Measure of all ML techniques. Moreover, while the results of DECA improve in Experiment II and Experiment III this does not happen for all the considered ML techniques. Indeed, their performances are quite stable along all the experiments, even if the training set grows up with the number of experiments with a precision and recall never exceeding the threshold of 38.3% and 26% respectively.

Summary RQ2: *DECA outperforms traditional ML techniques in terms of recall, precision and F-Measure when classifying e-mail content. Moreover, while the results of DECA improve in Experiment II and Experiment III this does not happen for all the considered ML techniques.*

V. DECA IN A REAL-LIFE APPLICATION: CODE RE-DOCUMENTATION

In this section we show how DECA can be used for a specific application, namely mining source code documentation from developers' communication. Specifically, a previous work by Panichella *et al.* [45] proposed an approach, based on vector space models and ad-hoc heuristics, able to automatically extract, with high precision (up to 79%), method descriptions from developers communications (bug tracking systems and mailing lists) of two open source systems, namely Lucene and Eclipse. The limit of such approach is that it tends to discard a high number of potentially useful method descriptions. Indeed, the approach discarded around 33% and 22% useful paragraphs for Lucene and Eclipse respectively. However, the authors pointed out that several are the *discourse patterns* that characterize *false negative method descriptions*.

We argue that DECA can be successfully used to overcome such limitations capturing some of the *discourse patterns* contained in *false negative method descriptions*. Thus, we experimented our *intention* based approach on all the 200 paragraphs (100 for each project) validated in [45] to provide a preliminary evaluation of how many (useful) paragraphs (i.e., false negatives) could be recovered by using our approach. Specifically, we considered as valid method descriptions the paragraphs containing for DECA sentences belonging to *Feature Request, Problem Discovery, Information Seeking* or *Information Giving* categories (in according to the taxonomy defined in Section II-A) because are more likely to contain information about the *behaviour* of a Java method.

Table X reports for each project: (i) the number of analyzed paragraphs, (ii) the number of paragraphs previously labeled as false negatives (FN), (iii) the number of FN recovered by DECA, and (iv) the number of false positives (FP) generated by DECA. For Eclipse, DECA was able to recover about 64% of paragraphs previously labeled as false negatives, while for the Apache Lucene DECA recovered about 79% of them. Moreover, DECA generates a reduced set of false positives for both projects achieving a precision of 74% for Eclipse and 70% for Lucene. These results demonstrate how DECA can improve significantly the recall of the previous approach, even if we obtain a slight degradation of the performance in terms of precision.

Table XI shows some examples of paragraphs correctly detected by DECA that were marked as false negatives in the previous work by Panichella *et al.* [45]. This happens because the previous approach assigned a score to paragraphs to be candidate method documentation if they contain some keywords, such as, "return", "override", "invoke", etc. However, there can be valid method descriptions not containing anyone of such keywords. As a consequence, these paragraphs were discarded by the approach of Panichella *et al.* [45]. Instead, as it can be noticed from the examples reported in Table XI, DECA is able to identify at least 70% of them. For example, in the case of Eclipse, the paragraph referring to the *build* method from the *JavaBuilder* class contains the sentence *"JavaBuilder.build() triggers a PDE model change..."*. DECA successfully recovers such paragraph and correctly assign this method description to the *Information Giving* category. A similar situation occurs for the others recovered paragraphs.

Thus, *intention mining* performed by DECA could improve the recall (and precision) of a previous approach that mine source code documentation from developers communication means. Specifically, this result shows that DECA really overcomes limitations of traditional lexicon approaches (e.g. as LDA) that are not able to capturing capturing *discourse patterns* contained in paragraphs useful for code re-documentation.

TABLE X: Number of paragraphs recognized by DECA

Project	# of paragraphs	# of FN generated by the previous approach	# of FN recovered by DECA	# of FP generated by DECA
Eclipse	100	22	14	5
Lucene	100	33	26	11

TABLE XI: Examples of paragraphs recognized by DECA

Project	Class	Method	Paragraph
Eclipse	JavaBuilder	build	JavaBuilder.build() triggers a PDE model change, which causes classpath computations and more builds to happen. The build performs validation of build.properties files, adding markers for various warnings. The marker change then triggers the above code again, triggering a plugin model change.
Eclipse	FileSystemResourceManager	build	The FileSystemResourceManager.delete operation stays at 0% and then after a minute jumps to 99%
Lucene	NearSpansUnordered	skipTo	The problem with skipTo() and next() is that are not designed to be used both in the same search. As the doc of the scored Spans is normally ahead, ... NearSpansUnordered adds a check in skipTo() for the scored spans not being ahead of the target of skipTo().
Lucene	TrecFTParser	parse	In TrecFTParser.parse(), you can extract the logic which finds the date and title into a common method which receives the strings to look for as parameters (e.g. find(String str, String start, int startlen, String end)).

VI. THREATS TO VALIDITY

Threats to internal validity concern any confounding factor that could influence our results. This kind of threats can be due to a possible level of subjectivity caused by the manual classification of entities. To reduce this kind of threats we attempt to avoid any bias in the building of oracles, by keeping one of the human judges who contributed to define our oracle tables unaware of all defined and implemented patterns. Moreover, we built the oracle tables on the basis of predictions that two human judges separately made. Only predictions, which both judges agreed upon, formed our oracles for the experiments. Another threat to internal validity could regard the particular ML classification algorithm used as baseline for estimating our results, as the results could be dependent on the specific technique employed. For mitigating this threat we used different ML algorithms and compared the results achieved through our approach with the results obtained through each of them.

Threats to external validity concern the generalization of our findings. In our experiments, we used a subset of messages in the original mailing lists. This factor may be a threat to the external validity, as the experimental results may be applicable only on the selected messages but not to the entire

mailing lists. To reduce this threat, we tried to use as many messages as possible in our experiments. For the same reason we prepared different test sets in which we tried to select both messages related to different periods of the same year and messages related to different development stages. Messages posted in the same month often belong to the same discussion threads and often include quotations of messages to which they reply. To avoid analyzing more times the same sentences, for the experiments II and III we randomly selected test sets containing messages posted in time windows of five months. Another threat to the external validity is represented by the mailing lists used in our experiments. It is possible that some particular characteristics of mailing lists we selected lead to our experimental results. To reduce this threat we used two different existing development mailing lists (as we discussed in Section III-B). Moreover we further experimented our approach with text fragments belonging to mailing lists of two others open source projects (Eclipse and Lucene). However, in the future our aim is to assess our approach on mailing lists of more projects of different nature (both industrial and open source) asking developers to validate the sentences according to the categories defined in this paper.

VII. Related Work

In the last years, several research works proposed various approaches based on NLP analysis as tools to derive important information aimed at supporting program comprehension and maintenance activities [22], [40], [29], [42], [57], [55], [33], [54]. Recent studies used Natural Language Parsing to classify the content of natural language text in according to pre-defined categories.

Shihab *et al.* [50] showed that mailing list discussions are closely related to source code activities and the types of mailing list discussions are good indicators of the types of source code changes (code additions and code modifications) being carried out on the project. In the literature several techniques have been presented to speed up and facilitate managing emails. The work by Corston-Oliver *et al.* [16] 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.

Ko *et al.* in [34] presented a study that observed (1) how noun, verbs, adverbs and adjectives are used to describe software problems; (2) to what extent these parts of speech can be used to detect software problems. We share with this work the idea of using Natural Language Parsing to detect recurrent patterns in descriptions reported by developers (in our case in mailing list data) allows to automatically classify the content written in natural language text, relevant for development tasks.

Cohen *et al.* in [15] presented an approach that relying on machine learning methods classifies emails according to the intent of the sender. Differently from these previous works, we focus our attention in classifying emails sentences written by developers during development discussions (in mailing lists data). Thus, we analyzed syntactic and semantic information to discover significant recurrent patterns useful to recognize relevant sentences within messages. It is important to highlight that sentence classification has been used in several practical application domains, including analysis of biomedical data, [39], [56], generation of scientific summaries based on textual analysis [31], [51], legal judgment [28], product reviews [52], customer complaints in help desks [32].

VIII. Conclusions and Future Work

In this paper we presented DECA (Development Email Content Analyzer), an approach based on natural language parsing to automatically classify the content of development communication (e.g., emails) according to likely developer intentions, such as asking/providing helps, proposing a new feature or reporting/discussing a bug. The approach builds on top of an English natural language parser and applies a set of heuristics that have been defined on a training set.

To evaluate DECA, we have carried out an empirical study on 202 emails from the QT project and 100 emails from the Ubuntu Linux distribution. Results for the study indicates that DECA achieves high levels of precision (90%) and recall (about 70%), outperforming a classification performed with six different machine learning techniques. Also performances of DECA improve along three experiments in which we improved the heuristics by exploiting false positives and false negatives of the previous experiments, which a rate of improvement varying between 31% and 58%.

We also showed that our approach can be used to mine source code documentation from developers communication means (see Section V), and that it improves the recall of a previously preposed approach [45] based on vector space models and heuristics. Thus, as a first direction for future work, we plan to implement specific heuristics to detect paragraphs useful to re-document Java methods (classes) performing a larger study on data from different communications means such as, mailing lists, issue trackers and IRC chats.

The proposed approach can be used for a wider application domain, such as the preprocessing phase of various summarization tasks. For example, DECA could be used as a preprocessing support to discard irrelevant sentences within emails or bug report summarization approaches [5], [36], [46], [48]. Furthermore, we plan to use DECA in combination with topic models for retrieving contents presenting the same *intentions* and treating the same topics from developers discussions. For example, such a combination could enable the possibility for a developer to retrieve all feature requests related to a given topic from different communication means in order to plan a set of change activities.

Acknowledgment

Sebastiano Panichella gratefully acknowledges the Swiss National Science foundation's support for the project *"Essentials"* (SNF Project No. 200020−153129). We also thank the reviewers of this paper for the very insightful comments useful to improve our research work.

REFERENCES

[1] J. Aranda, and G. Venolia, *The secret life of bugs: Going past the errors and omissions in software repositories*. In Proceedings of the 31st International Conference on Software Engineering (ICSE), 2009, pp. 298-308.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y. Guhneuc, *Is it a bug or an enhancement?: a text-based approach to classify change requests*. CASCON, 2008:23.

[3] J. Anvik, L. Hiew, and G.C. Murphy, *Who should fix this bug?*. In Proceedings of the 28th International Conference on Software Engineering (ICSE), 2006, pp. 361-370.

[4] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. *Content classification of development emails*. In Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 375-385.

[5] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, *CODES: mining source code description from developers discussions*, in Proceedings of the 22th IEEE International Conference on Program Comprehension (ICPC), 2014, pp. 106-109.

[6] R. A. Baeza-Yates, B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[7] V. R. Basili, L. C. Briand, and W. L. Melo, *A validation of object oriented design metrics as quality indicators*. In IEEE Trans. Software Eng., vol. 22, no. 10, 1996, pp. 751-761.

[8] A. Begel and N.Nagappan, *Global Software Development. Who Does It?*. In Proceedings of the 2008 IEEE International Conference on Global Software Engineering (ICGSE), 2008, pp. 195-199.

[9] M. Bezerra, A. L. I. Oliveira, and S. R. L. Meira, *A constructive rbf neural network for estimating the probability of defects in software modules*. In Neural Networks, 2007. IJCNN 2007. International Joint Conference on, 2007, pp. 2869-2874.

[10] D. M. Blei, A.Y. Ng, and M. I. Jordan, *Latent dirichlet allocation*. In Journal of Machine Learning Research (JMLR), Vol. 3, 2003, pp. 993-1022.

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

[12] D. Cer, M.C. de Marneffe, D. Jurafsky, and C. D. Manning, *Parsing to Stanford dependencies: Trade-offs between speed and accuracy*. In Proceedings of the 7th International Conference on Language Resources and Evaluation (LREC), 2010.

[13] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora, *A Hidden Markov Model to detect coded information islands in free text*. In Proceedings of 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 157-166.

[14] E. Ceylan, F. Kutlubay, and A. Bener, *Software defect identification using machine learning techniques*. In Software Engineering and Advanced Applications, 2006. SEAA 06. 32nd EUROMICRO Conference on, 2006, pp. 240-247.

[15] W. W. Cohen, V. R.Carvalho, and T. M. Mitchell, *Learning to Classify Email into "Speech Acts"*. In Proceedings of Empirical Methods in Natural Language Processing, 2004, pp 309-316.

[16] S. Corston-Oliver, E. Ringger, M. Gamon, and R. Campbell, *Task-focused summarization of email*. In Proceedings of the ACL Workshop Text Summarization Branches Out, 2004, pp 43-50.

[17] I. Dagan, O. Glickman, and B. Magnini, *The PASCAL recognizing textual entailment challenge*. In Proceedings of the First International Conference on Machine Learning Challenges: evaluating Predictive Uncertainly Visual Object Classification, and Recognizing Textual Entailment, 2005, pp. 177-190.

[18] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Fumas, R. A. Harshman, *Indexing by Latent Semantic Analysis*. In Journal of the American Society of Information Science, Vol. 41, No. 6, pp. 391-407, 1990.

[19] M.C. de Marneffe and C.D. Manning, *The Stanford typed dependencies representation*. In COLING 2008: Proceedings of the Workshop on Cross-framework and Cross-domain Parser Evaluation, 2008, pp 1-8.

[20] M.C. de Marneffe, and C.D. Manning, *Stanford dependencies manual*. Technical Report, 2008.

[21] M.C. de Marneffe, B. MacCartney, and C.D. Manning, *Generating typed dependency parses from phrase structure parses*. In Proceedings of the International Conference on Language Resources and Evaluation (LREC), 2006, pp 449-454.

[22] S. S. Deshpande, G. K. Palshikar, , and G. Athiappan, *An Unsupervised Approach to Sentence Classification*. In International Conference on Management of Data (COMAD), 2010, pp 88-99.

[23] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[24] K. Fundel, R. Küffner and R. Zimmer, *RelEx – Relation extraction using dependency parse trees*. In Bioinformatics, v.23, n.3, 2007, pp 365-371.

[25] B. Glaser, and A. Strauss, *The discovery of grounded theory: Strategies of qualitative research*. New York, NY:Aldine de Gruyter, 1967.

[26] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, *Communication in open source software development mailing lists*. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013, pp 277-286.

[27] A. Guzzi, A. Begel, J.K. Miller, and K. Nareddy, *Facilitating Enterprise Software Developer Communication with CARES*. In Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp 1367-1370.

[28] B. Hachey and C. Grover, *Sentence classification experiments for legal text summarization*. In Proceedings of 17th Annual Conference on Legal Knowledge and Information Systems (Jurix-2004), 2004, pp. 29-38.

[29] S. Hen, M. Monperrus, M. Mezini, *Semi-automatically extracting FAQs to improve accessibility of software development knowledge*. In Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE), 2012, pp. 793-803.

[30] K. Herzig, S. Just, and A. Zeller, *It's not a bug, it's a feature: how misclassification impacts bug prediction*. In Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 392-401.

[31] F. Ibekwe-Sanjuan, S. Fernandez, E. Sanjuan, and E. Charton, *Annotation of scientific summaries for information retrieval*. In O.A.H. Zaragoza, editor, ECIR08 Workshop on: Exploiting Semantic Annotations for Information Retrieval, 2008, pp. 70-83.

[32] A. Khoo, Y. Marom, and D. Albrecht, *Experiments with sentence classification*. In Proceeding of 2006 Australasian Language Technology Workshop (ALTW), 2006, pp. 18-25.

[33] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, *Enriching Documents with Examples: A Corpus Mining Approach*. In Journal of ACM Transacrions on Information Systems (TOIS), Vol. 31, Issue n. 1, January 2013, Article n. 1.

[34] A. J. Ko, B. A. Myers, and D. H. Chau, *A linguistic analysis of how people describe software problems*. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2006, pp 127-134.

[35] P. S. Kochhar, Tien-Duy B. Le, and D. Lo, *It's not a bug, it's a feature: does misclassification affect bug localization?*. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 296-299.

[36] D. Lam, S.L. Rohall, C. Schmandt, and M.K. Stern, *Exploiting e-mail structure to improve summarization*. In Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW), Interactive Posters, New Orleans, LA. 2002.

[37] T. D. LaToza, G. Venolia, and R. DeLine, *Maintaining mental models: a study of developer work habits*. In Proceedings of the 28th International Conference on Software Engineering (ICSE), 2006, pp. 492-501.

[38] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, *Evolutionary optimization of software quality modeling with multiple repositories*. In IEEE Trans. Softw. Eng., vol. 36, no. 6, 2010, pp. 852-864.

[39] L. McKnight and P. Srinivasan, *Categorization of sentence types in medical abstracts*. In Proceedings of American Medical Informatics Association Annual Symposium, 2003, pp 440-444.

[40] W. Maalej and M. P. Robillard, *Patterns of Knowledge in API Reference Documentation*. In IEEE Trans. Software Eng. 39, no.9, 2013, pp 1264-1282.

[41] J. Nivre, L. Rimell, R. McDonald, and C. Gómez-Rodríguez, *Evaluation of dependency parsers on unbounded dependencies*, in Proceedings of COLING, 2010, pp. 813-821.

[42] R. Pandita, X.Xiao, H.Zhong, T.Xie, S.Oney, and A.Paradkar, *Inferring method specifications from natural language API descriptions*. In Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE), 2012, pp 815-825.

[43] S. Panichella, M. Di Penta, and G. Canfora. *How the Evolution of Emerging Collaborations Relates to Code Changes: An Empirical Study*. In Proceedings of 22nd International Conference on Program Comprehension (ICPC), 2014, pp 177-188.

[44] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol. *How Developers Collaborations Identified from Different Sources Tell us About Code Changes*. In Proceedings of 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 251-260.

[45] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, *Mining source code descriptions from developer communications*. In Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC), 2012, pp. 63-72.

[46] O. Rambow, L. Shrestha, J. Chen, and C. Lauridsen, *Summarizing email threads*. In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL) Short paper, 2004, pp 105-108.

[47] S. Rastkar, G. C. Murphy, and G. Murray, *Automatic Summarization of Bug Reports*. In IEEE Transactions on Software Engineering, Vol. 40, Issue 4, 2014, pp. 366-380.

[48] S. Rastkar, G. C. Murphy, and G. Murray, *Summarizing software artifacts: a case study of bug reports*. In Proceedings of 32nd International Conference on Software Engineering (ICSE), 2010, pp 505-514.

[49] P. C. Rigby, and A. E. Hassan, *What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List*. In Proceedings of the 4th International Workshop on Mining Software Repositories, 2007, page 23.

[50] E. Shihab, N. Bettenburg, B. Adams, and A. E. Hassan, *On the central role of mailing lists in open source projects: an exploratory study*. In Proceedings of the 2009 International Conference on New Frontiers in Artificial Intelligence (JSAI-isAI), 2009, pp. 91-103.

[51] S. Teufel and M. Moens, *Sentence extraction and rhetorical classification for flexible abstracts*. In AAAI Spring Symposium on Intelligent Text summarization, Stanford, 1998, pp. 16-25.

[52] C. Wang, J. Lu, and G. Zhang, *A semantic classification approach for online product reviews*. In Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence, 2005, pp. 276-279.

[53] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[54] E. Wong, J. Yang, and L. Tan, *AutoComment: Mining question and answer sites for automatic comment generation*. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 562-567.

[55] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, *Automated Extraction of Security Policies from Natural-Language Software Documents*. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE), 2012, pp 12:1-12:11.

[56] Y. Yamamoto and T. Takagi, *Experiments with sentence classification: A sentence classification system for multi biomedical literature summarization*. In Proceedings of 21st International Conference on Data Engineering Workshops, 2005, pp. 1163-1168.

[57] H. Zhong, L. Zhang, T.Xie, and H.Mei, *Inferring resource specification from natural language API documentation*. In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2009, pp. 307-318.

[58] Y. Zhou, Y. Tong, R. Gu, and H. Gall, *Combining Text Mining and Data Mining for Bug Report Classification*. In Proceedings of 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 311-320.

[59] T. Zimmermann and N. Nagappan, *Predicting defects with program dependencies*. In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, 2009, pp. 435-438.