On The Road to HADES–Helpful Automatic Development Email Summarization

Alberto Bacchelli, Michele Lanza, Ebrisa Savina Mastrodicasa
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—During the development of software systems, programmers often discuss issues such as structural decisions, defects, time management, *etc.* Especially in distributed development, where most of the discussions among developers take place via email, developers receive daily dozens, if not hundreds, of messages, either personally sent to them or broadcasted by mailing lists they are subscribed to. Dealing with these emails requires developers to spend long time reading voluminous amounts of text.**

A way to solve this problem is *summarization*: Producing a summary of a text means reducing it to a version based only on fundamental elements, by removing the parts unnecessary for the comprehension. Since researchers presented techniques for automatically summarizing natural language documents or source code, we are interested in investigating whether it would be possible to also summarize development emails, so that developers could consult small summaries instead of long threads. In this paper, on the basis of our first experiences in tackling development email summarization, we discuss the challenges that go hand in hand with such an endeavor.

I. MOTIVATION

Software development projects, especially if large-scale, require a remarkable coordination effort amongst the project workers. Given the intensity and problems that come with coordination, it is no wonder that coordination is considered the main reason why adding more developers to an ongoing software project might not speed up development [4].

Coordination is driven by communication. In co-located development teams, unplanned informal face-to-face meetings are the favorite form of communication when developers need to coordinate or face program comprehension problems [12]. Personal meetings, besides disrupting developers' attention and retaining knowledge by a few developers [10], are inapplicable to distributed development projects. Developers, thus, replace face-to-face meetings with electronic communication means. Instant messaging, wikis, forums are viable options, but the decisive role is played by emails, indeed: "Mailing lists are the bread and butter of project communications" [6]. Emails are asynchronous, thus evade time zone barriers and do not disrupt developers's attention; mailing lists broadcast discussions, announcements, and decisions to all participants, thus maintaining developers' awareness; emails are not bound to specific abstraction levels (as opposed to commit messages, design documents, or code comments), thus they can be used to discuss issues ranging from low-level decisions (*e.g.,* implementation, bug fixing) up to high-level considerations (*e.g.,* design rationales); mailing lists archive messages, thus offering a historical perspective.

Considered the numerous reasons why a programmer might start an email conversation with a single colleague or the entire rest of the team, and that a software project might be carried out for several months or years, it is easy to picture the vast amount of produced emails. For example, on the Apache developer mailing list, there were about 4,996 messages in the year 2004 and 2,340 in 2005, and for gcc, these numbers were 19,173 and 15,082 [3]. To maintain team coordination and perform work on a software system, software developers must keep track, read, and understand a voluminous amount of electronic communication. This is a challenge, not only because of the large number of emails–which require time and effort to be read–popping up day after day in a developer's mailbox, but also because retrieving past information from such messages can be time consuming and frustrating. Sometimes, the amount of information may be overwhelming, causing messages to be left unread and searches to be abandoned, which in turn lead to duplicate, uncoordinated, or non-optimized work to be performed [15].

One way to alleviate this information overload issue is to provide a summary of development emails. A correct and helpful summary (*i.e.,* the text reduced to a version without the parts unnecessary for the comprehension) enables developers to reduce the time spent reading new emails or perusing messages that have been returned from searches, found through browsing, or recommended by team members.

Given the growing amount of produced and recorded textual information, often referenced as *big data*, there is substantial interest and a large body of work in the automated generation of summaries for natural language documents, and software development related artifacts (*e.g.,* [7], [15]). On the basis of the success of existing efforts on automatic summarization, we wonder whether it would be possible to devise a tool, which we call HADES, to automatically generate helpful summaries of development emails. In this paper, we specifically analyze and discuss the challenges that go hand in hand with such an endeavor. We investigate the challenges in devising a successful research methodology and in creating an effective summarization technique to be implemented in a tool that can be used in real world scenarios.

Structure of the paper: In Section II, we present existing and related work in the area of textual summarization; in Section III, we present the challenges that have to be tackled, both in terms of research method and in creating an effective approach; in Section IV, we discuss our findings from a more high level perspective and conclude.

II. Related Work

Automatic summarization of documents written in natural language has been attempted by researchers for more than half a century [8]. We focus on efforts regarding summarization of emails and software development artifacts.

Emails constitute a composite, chronologically ordered, and structured (due to message metadata such as author, subject, or date) material that can be summarized with different methods. Lam *et al.* [11] proposed a system to summarize email messages by exploiting thread reply chains and commonly found features (like the proper names in subjects found in the main sentences of the body). They asked four participants to use their system on their own messages and provide feedback on the results. The gathered opinions were that the summary quality was either excellent or very bad (without intermediate results), the system worked best on either very short or very long email thread (sometimes, for short messages, reading the summary took longer), and the system would have been useful to decide which email to read first but it did not obviate the need to read it. Rambow *et al.* [14] proposed a technique to extract sentences exploiting machine learning techniques. By considering a thread as a single text, they fixed an incremental set of features that composed a vector view of every sentence in the thread itself. They computed features that are usable for all text genres (*e.g.,* the length of the sentence), features obtained considering the thread broken into messages (*e.g.,* absolute position of a message in the thread), and features specific to the thread structure (*e.g.,* number of responses to a message, number of recipients of a message). They presented results in terms of precision and recall with respect to a golden set of sentences selected by two human readers, but did not evaluate resulting summaries using word/string based similarity metric and/or human judgments. Other summarization efforts include the one of Carenini *et al.*, who built a fragment quotation graph to represent the structure of the conversation and subsequently propose the usage of clue words (*i.e.,* words that appear in both parent and child messages), to improve the quality of the summarization [5].

Concerning software artifacts, Rastkar *et al.* investigated an approach to summarize issue reports [15]. They found that existing conversation-based extractive summary generators can produce summaries for reports that are better than a random classifier, especially if trained on issue report data. To evaluate reports they asked human judges, who agreed that the generated extractive summaries contain important points from the original report and are coherent. Haiduc *et al.* proposed a technique for automatically summarizing source code, leveraging the lexical and structural information in the code [7]. Their preliminary study showed that we can use text retrieval methods for this problem and the results are better than the state of the art in natural language summarization.

III. Challenges

After analyzing the related work in document and software artifact summarization, we started to investigate whether it could be possible to automatically generate helpful summaries of development emails. By exploring possible research methods and approaches to devise an effective technique, we found many unanswered questions while performing this task. In the following we analyze these open questions, so that we can see the state-of-the-practice in this topic and underline aspects that call for further studies.

A. Methodology

Given the same development email, distinct people are very likely to summarize it differently and give differing feedback on others' summaries. This is true not only in the case of *abstractive* summaries (*i.e.,* composing a document with new sentences that express only the core messages), but also for *extractive* ones (*i.e.,* selecting a subset of existing sentences to form the summary). This situation binds the results of any summarization approach to the subjective evaluation of humans. To alleviate this issue, we have to follow a sound research method that considers the twofold nature (*i.e.,* quantitative and qualitative) of this topic. Unfortunately, there is no well-established research method: Researchers adopted dissimilar approaches in their studies for both creating golden sets of summaries and evaluating the output of their approaches. Especially in the software engineering domain, where only few studies cover artifact summarization, the proposed research methods appear scattered and inconsistent with one another. For this reason, the first open challenge in this topic is finding a common base for a sound research method.

We claim that the two main points when reasoning about the research method to use are the development email *corpus* and the *participants* to be involved in the study.

Corpus: Methodological questions regarding the corpus start by deciding the population from which we can extract the sample development emails. First, it is not clear yet whether emails written in open source system (OSS) communities are equivalent to those written in industrial settings, in terms of summarization features. Could we generalize findings about summarization learnt from OSS emails to other settings? Second, developers write email both personally, one-to-one, to other colleagues and publicly to mailing lists for broadcasting. In this case, the main difference is the number of participants having a role in the discussion. How does this difference impact summaries? How much additional value do we get with summarization features considering participants? In addition, manually summarizing emails and manually evaluating automatically generated summaries take time. This limits the size of corpora of summarized messages. Yet, we might assume that some of the features appropriate for summarization can be learnt from a reasonably small sample of well chosen examples. How can we compute the

size of the smallest, yet representative, email sample? Should we have a smaller sample, analyzed by many annotators per email, or a larger sample, with fewer annotators per email?

Participants: Finding the proper participants for software engineering research efforts is a daunting task, especially when research is conducted in an academic setting. The challenge regards *expertise*, *disagreement*, and *task difficulty*.

- **Expertise**: To evaluate approaches for automatically summarize general purpose emails, researchers have been employing corpora such as the Enron dataset [9]. This kind of emails can be almost totally understood by non-experts, who can easily generate the summaries. On the other hand, software artifacts, such as issue reports and development emails, contain technicalities that require more expertise to be correctly interpreted; for this reason, researchers have generally employed participants with a computer science background, mostly graduate students (*e.g.,* [7], [15]). However, how correctly can non-experts in a specific software project capture the proper meaning of emails, especially those referring to implied parts or using jargon? Rastkar *et al.* explain that "summaries created by experts might rely on knowledge that was not in the bug reports, potentially creating a standard that would be difficult for a classifier to match." Should we have the *best* golden set, or is the most reasonable and simple one to create enough?
- **Disagreement**: Due to the subjectivity of the summarization task, researchers reported medium-low to very low agreement (mostly less than 0.4 value in the kappa test, *e.g.,* [15]) in summaries extracted or generated by human participants. An open challenge is deciding what value of agreement is "good enough" for software engineering research and how we can handle this disagreement to manage or reduce it. What are the points of disagreement? Do we achieve better agreement when the participants are more expert of the systems being discussed in the emails? What will the impact on data reliability be, if we ask participants to collaborate and reach a final agreement on their initial summaries?
- **Task Difficulty**: Participants in summarization studies are mostly asked to perform two different tasks: Generating a summary of a given document and evaluate the quality of a summary already produced. These tasks require a different effort: At least time wise, evaluating a given summary seem to be less problematic. Even though human generated summaries might be also used to validate automatic generated ones, their main usage is to *learn* features to be implemented by algorithmic techniques. Would it be right to let researchers analyze and realize what the best summarization features are and just ask participants to rate automatically generated summaries? How strong is the risk that participants may just want to please experimenters while evaluating?

B. Abstractive or Extractive?

Our aim is to automate the summarization process and obtain emails that emulate the characteristics of the original ones, without losing the meaning. Summaries can be either *abstractive* or *extractive*. An extractive summary contains a subset of the text belonging to the original email. Since with this method the text is extrapolated from their original context, reordering may be needed to make the summary comprehensible. Abstractive summarization refers to composing a document with new sentences that contain the core messages of the original one. This second form produces more fluent summaries as it respects natural language semantic rules. The current state-of-the-art in abstractive techniques has not yet supported meaningful application, while extractive summaries present a number of advantages, such as:

- An extractive process is more lightweight than an intelligent procedure of summary composition. This translates into a reduced computation time.
- By using entire parts of the text included in the original email, it is impossible to compose new phrases with incorrect synonyms. Even if the flow between parts might result shaky, the internal meaning of every single part remains the same.
- When users read some text in the summary, they can easily link it back to the original email if needed. On the contrary, tracing back a topic from an abstractive summary to the original email requires more time.

Despite this situation, given the small work on summarization in the software engineering domain, we believe that choosing the best approach still remains an open question.

C. Keywords or Sentences?

By deciding to perform extractive summaries, at least because of their technical advantages, we are facing another question: How much and which kind of text should we extract from development emails? In related work, we find two alternative portions of text used for extraction: keywords or sentences. In the current-state-of-practice, keyword extraction mainly finds quantitative application in text mining [2] (*e.g.,* trend and event detection in stream of documents, document clustering, or creation of tag cloud visualizations), while sentence extraction is used when the final output is read by people. In the case of source code summarization, the choice is keywords, mainly processed function names or identifiers.

Similarly to previous choices, there is no widespread adoption of an extractive approach in software engineering. For this reason, we started investigating this topic ourselves [13], by involving two undergraduate students in informatics. We performed a small pilot study where we asked the participants to (1) read 6 six email threads chosen from the ArgoUML mailing list, (2) summarize three threads extracting keywords and three extracting sentences, and (3) answer a few debriefing questions about the difference when summarizing

with keywords and with sentences. We also recorded the time necessary for producing the summaries. In case of participant *A*, producing keyword summaries required double the time (compared to sentence summaries), while the opposite held for participant *B*. Participant *A* found it easier and more natural to extract sentences, because she had the feeling that by extracting keywords there was a high risk of missing relations among them. On the contrary, participant *B* found keyword extraction easier and more appropriate; indeed he spent less time in this summarization technique. In the case of keywords, both the participants felt the need to extract not only single terms but *n*-grams composed of 2 to 4 words, especially with bi-grams such as "not correct."

Even though this was a small pilot study, we found this experience interesting: Analyzing the answers of two students with exactly the same background in informatics, it is still not clear whether keyword or sentence extraction should be used for automatically summarize email threads. This situation calls for a more in-depth study of advantages and drawbacks of summarization approaches both from the perspective of study participants and from that of final users.

D. Not Only Natural Language

Most of the general purpose summarization approaches are tested on well-formed, or sanitized, natural language documents. When summarizing development emails, however, we have to deal with natural language text which is often not-well formed and is interleaved with languages with different syntaxes, such as code fragments, stack traces, patches, *etc.*

Currently no summarization technique takes this aspect into account, for example Rastkar *et al.* explained that they intentionally avoided issue reports with these features: "We avoided selecting bug reports consisting mostly of long stack traces and large chunks of code as this content may be used but is not typically read by developers" [15]. On the contrary, we claim that the presence of these parts written in different languages are the most domain specific feature of the summarization of development emails, and software artifacts in general; we state that not considering this aspect can undermine the effectiveness of the proposed methods [1]. Moreover, the presence of different languages should be taken into account when considering the size and the element in the email sample to construct benchmarks and golden sets.

E. Meta-modeling Email Threads

The email thread as a whole is a collaborative effort with interaction among the discourse participants. Since replies do not happen immediately, the responses need to identify relevant elements of the discourse context (*e.g.,* by citing previous messages). Researchers presented techniques for using the intrinsic characteristics of email threads (*e.g.,* repetitions of parts) to create better general purpose summaries [5]. However, trying to replicate their approaches on development mailing list emails, we encountered a challenge: What is the most appropriate *meta-model* for email threads and their messages, so that we can derive precise features for each sentence? An elementary solution would sort emails in a thread chronologically, removing quoted text in the replies. However, this would imply losing the binding among the different parts. On the other hand, reconstructing the model of a thread is not a trivial task, because of the noisy nature of emails, where text is not correctly and entirely repeated in the right places (*e.g.,* due to 80 character line limitation). Moreover, there are different posting style when replying to emails: interleaved posting (also know as inline replying, where relevant parts in the original message are quoted), and bottom-posting and top-posting (where the reply follows and precedes the original message, respectively). If interleaved posting is practical for reconstructing thread models, the other two cases leave open the question whether to consider the quoted text entirely or not. This requires further investigation.

F. Scoring Extracted Parts

Given that extractive summaries created by different people tend to significantly diverge, determining which of the parts chosen by annotators should be considered in the final golden set is a non-trivial task. Researchers mostly had an odd number of annotators per email, so that they could pick parts selected by the majority for the golden set. However, we deem that a part chosen by all the annotators should have more weight than parts chosen by just half of the participants. This requires the creation of a valid scoring system, which would also ease the replicability of studies on summarization techniques.

IV. Conclusion

Given the vast amount of communication related to software development that takes place through emails daily, and the subsequent information overload, we claim that the summarization of development emails would be a helpful contribution to the software engineering community; not only for reducing the amount of text to be read daily be developers, but also to improve retrieval of pertinent information from email archives.

In this paper we presented the numerous challenges and questions that are still open on this topic, and the next steps that researchers will have to face in the future to devise methods and tools for automatically generating helpful summaries of development emails. We found that not only precise and technical challenges are still open (*e.g.,* a scoring system for parts selected by annotators, a correct meta-model for email threads, or the analysis of parts not in natural language), but we are also missing a widespread and consolidated research method for devising these kinds of studies. Moreover, studies in related fields can guide only partially our future research on this topic, due to the scattered results and the many differences with the given domain.

References

[1] A. Bacchelli, T. dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *In Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385, 2012.

[2] M. W. Berry and J. Kogan. *Text Mining: Applications and Theory*. Wiley, 2010.

[3] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of MSR 2006 (3th International Workshop on Mining Software Repositories)*, pages 137–143. ACM, 2006.

[4] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 2nd edition, 1995.

[5] G. Carenini, R. T. Ng, and X. Zhou. Summarizing email conversations with clue words. In *Proceedings of WWW 2007 (16th International World Wide Web Conference)*, pages 91–100, 2007.

[6] K. Fogel. *Producing Open Source Software*. O'Reilly Media, first edition, 2005.

[7] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 223–226. ACM, 2010.

[8] K. S. Jones. Automatic summarising: The state of the art. *Information Processing and Management*, 43(6):1449–1481, 2007.

[9] B. Klimt and Y. Yang. Introducing the enron corpus. In *Proceedings of CEAS 2004 (1st Conference on Email and Anti-Spam)*, 2004.

[10] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 344–353. IEEE Computer Society, 2007.

[11] D. Lam, S. L. Rohall, C. Schmandt, and M. K. Stern. Exploiting e-mail structure to improve summarization. 2002.

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

[13] E. Mastrodicasa. Extractive summarization of development emails. Bachelor's thesis, University of Lugano, June 2012.

[14] O. Rambow, L. Shrestha, J. Chen, and C. Lauridsen. Summarizing email threads. In *In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL) Short Paper Section*, 2004.

[15] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *In Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 505–514, 2012.