Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 12, 2012

Extractive Summarization of Development Emails

Ebrisa Savina Mastrodicasa

Abstract

During the development of a project, programmers discuss problems, structural decisions, bugs, time management, etc. To do so, they use instant messaging, write on dedicated forums, and exchange emails that form threads that are also considered as a documentation of the project.
Often programmers need to recover information, like patches, links to external resources, and snippets. The problem with such situations is that frequently the information is contained in some email received a long time ago (so the programmer should search among old conversations to find the right one), or in a long email thread (so the programmer should read tens or hundreds of lines). Dealing with massive mailing lists is hard: Reading the entire mailing list to find a specific information is impractical.

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.

Our solution is aimed at emulating the way humans identify the essential sentences in a text. After having collected a golden set of human generated summaries, we perform machine learning to understand how to automate the extractive process of emails. We believe that identifying features frequently owned by sentences selected in human generated summaries is possible. We implemented a system which extracts sentences matching these recurrent features from an email, and creates a summary to be shown to the user.

Advisor
Prof. Michele Lanza
Assistants
Alberto Bacchelli, Dr. Marco D'Ambros

Advisor's approval (Prof. Michele Lanza): Date:

Contents

Acknowledgments

I would like to thank so much Prof. Michele Lanza and Dr. Marco D'Ambros to have been so helpful and kind: all of you made me understand that I have to believe more in myself. The most important concept I learnt during my bachelor project is:

"Don't think whether you are able to do it or not. Just do it!" - Michele Lanza.

I want to thank Lorenzo Baracchi, Lucia Blondel, Cristian Bruseghini, Mattia Candeloro, Andrea Gallidabino, Giulia Giacalone, Jacopo Malnati, Thomas Mantegazzi, Carlo Alberto Mastrodicasa, Wahed Abdul Mehran, Stefano Pongelli, Fabio Rambone, Giorgio Ratti, Marcello Romanelli, Igor Moreno Santos, Thomas Selber, Yotam Sharon, Giulio Valente. Without you all, I would have never be able to perform my research experiments; so this paper would not exist. Thank you!

A warm embrace goes to my family, my extraordinary boyfriend, my classmates and my friends. Thanks to have always supported me even in times of greatest stress.

A special thank goes to Alberto Bacchelli. You are an amazing person and I have been very lucky to walk this interesting path with you. Thanks to have always been there whenever I felt not so confident or I just needed to talk with a friend.

1 Introduction

1.1 Software engineering

When experts talk about "engineering", people think of new ideas; the design, the construction, and the sale of some product. This is true for most engineering fields like chemical, electrical, and mechanical engineering [3].

The situation is different for software engineering. Here the development process does not present any kind of physical construction of the product; the main development phases are Requirements Gathering, Analysis and Specification, Implementation, Testing, and Deployment [13]. Proper documentation should ideally complete each stage, to enable people involved in the following ones (or in future modifications of the system) to fully understand taken decisions, prior work, and problems they could encounter.
Requirements Gathering is the moment in which an analyst must understand what the customer is asking.
Analysis and Specification is the phase in which the manager communicates in detail to the developing team the information gathered from the customer; then software designers analyze the requirements and create an architecture of the system. Moreover they determine a schedule of the work. During these first steps the documentation is written by a person or a small group in a verbose form (in other words, in a form similar to this section).
Then the programmers of the team *implement* the system. In the *Testing* phase, the team verifies the features in the system using a set of test cases. The goal is to discover as many defects as possible and correct them, or to understand how to improve system performance.
Finally there is the *Deployment* of the application. It means that the application becomes available to the users. Usually, however, this is not the end: It often happens that the team has to perform many cycles of these phases before obtaining the desired result.

In the *Implementation and Testing* phases, the documentation form can change radically. Since programmers exchange information also by chatting in long email threads, if this material is topically and chronologically catalogued, it might be useful as additional documentation [1]. For this reason, often mailing lists are integrated in the documentation of a project [10].

1.2 Relevance of summaries

There exist many reasons why a programmer starts an email conversation with the rest of the team: Asking suggestions, sharing a discovered bug with the rest of the team, sharing useful information gathered from external resources (web, experts in the field, conferences), communicating new ideas about how to evolve the system, and deciding who must write the documentation. Considering this certainly not exhaustive list and the fact that a project is carried on for months or years, it is easy to imagine the vast amount of produced emails.

One of the main problems of mailing lists is that they are useful while the conversation is open and active, because every group member is notified about incoming replies and can dispose immediately of all the new information. However, when a conversation has been abandoned for some time, going back and searching for specific information becomes difficult and time consuming.

By considering these issues, we come to **summarization** as possible solution. A summary is useful only if it is significantly shorter than the original text, the grammatical rules are respected, and the sequence of sentences allows a natural and quick understanding. The human generated ones usually have these characteristics.
Talking strictly about mailing lists, a good human summary should contain for example the main concepts regarding the subject of the email, class names, web links, method signatures, first lines of traces, etc. A tool able to produce similar summaries would save much time to programmers, since essential information of an email would immediately stand out.

1.3 Proposed solution: extractive summarization

Our aim is to automate the summarization process and obtain texts that emulate the characteristics of the humans' ones.

A summary can be either **extractive** or **abstractive** [14]. An extractive summary contains a subset of the sentences (or words) belonging to the original text. Since with this method sentences/words are extrapolated from their original context, reordering may be needed to make the summary comprehensible. This technique does not assume the machine to understand the real meaning of what it is summarizing. When we talk about abstractive summarization, we refer 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, but putting a machine

in charge of some "creative" production is always error prone, because the possibility of receiving a text with wrong concepts is high.

The solution we propose is: **Extracting the most relevant sentences from an email, aggregating them in a logical order, and returning this text to the user**. This choice stems from the following reasons:

- Since we use parts of the sentences included in the original email, composing new phrases with gibberish synonyms is improbable. Even if the logic among sentences might miss something, we are sure that the internal meaning of every single sentence remains exactly what it is supposed to be.
- An extractive process is more lightweight than an intelligent procedure of summary composition. This is very frequently translated in less time needed to get the result.
- When a user reads some sentence in the summary, he can quickly find it in the original email if needed. On the contrary, if he has a summary composed of different words (but same concepts) and wants to recover the exact point in the email where that topic was treated, he has to go through the entire conversation.

1.4 Structure of the document

In Section 2 we analyze some documents treating summarization also in other fields like bug reports, speeches, etc. In Section 3 we explain the several steps we scale to create a benchmark to be used in machine learning. In Section 4 we described the systems we implemented to summarize emails with sentence extraction technique. In Section 5 we briefly analyze the quality of the summaries we managed to automatically create with our system, and we propose some optimizations that could be developed in the future.

2 Related work

Summarization techniques proposed in the literature differ from each other in the type of material they take as subject. We analyzed a consistent group of documents including summarization of bug reports, source code, (non-development) emails, and spoken speeches. This step is important because allows to gather as many techniques as possible even from different research fields, and see whether some could be applied to our domain.

2.1 Bug Reports

A bug report is a document where every problem with the implementation is entirely recorded, starting from its arising, passing though the generated team discussions, and ending with the agreed solution.

Being quick and efficient at perusing a report while searching some specific artifact is desirable for a programmer because it saves precious time to devolve to the implementation. To achieve this goal, *Rastkar et al.* [11] considered an extractive approach and use 36 bug reports taken from 4 open-source projects (Eclipse Platform, Gnome, Mozilla and KDE). They:

1. selected a spurious group of expert and non-expert annotators,
2. assigned each one to a subset of the test corpus (3 annotators for each report),
3. asked them to write an abstractive summary (composed of 250 words at most) for each given bug report,
4. asked them to link every sentence of their summary to one or more sentences in the original text,
5. selected a set of 24 features extracted from the summaries and computed the F-score value for each feature [1].

They have been able to create summaries with "*reasonable accuracy*". Nevertheless they were aware that a part of the used set of annotators might have written the required summaries without really understanding the meaning of the original text, since they were non-experts. This fact may have added some inaccuracy to the phase of features analysis.

From this document, we consider valid choices the extractive approach and the use of annotators to create a golden set of human generated summaries.

2.2 Source Code

The capacity of quickly reading and comprehending a long piece of source code is a dream for every programmer. Usually they approach this activity either by reading only methods' signatures, or by going through the whole code [6]. The first can be not sufficient to completely understand the piece of code. The second is time consuming (sometimes even impracticable). Therefore, as *Haiduc et al.* [6] reported, the best solution is something intermediate, which reflects the original structure and natural developers' understanding of the code.

Since a piece of source code is written following rigid rules and there can be no subjective interpretation with respect to its behavior, their approach was extractive. Their aim was to show that text retrieval techniques can be applied to code summarization. So they proceeded as follows:

1. They performed stop-words (English words so common as to be insignificant) removal and stemming (replace declined/conjugated words with their root).
2. They set up a case study involving 6 graduate students expert in Java programming.
3. They chose the open source audio player ATunes[2] as tool for the test.
4. The task of the students was to produce a summary for 12 methods taken from the same source code entity (document in the ATunes corpus), by selecting 5 terms from a list of terms present in the original source code. The highlighted ones should have described the method in the most appropriate way.

For what concerns the generation of automated summaries, they used LSI technique[3], and compute the cosine distance (used as score) between every term in the original methods and the ones in the test outputs. For each

[1] F-score is a measure of the accuracy of a test computed by considering *precision* (the number of correct results divided by the number of all returned results) and *recall* (the number of correct results divided by the number of results that should have been returned.) [12]

[2] `http://www.atunes.org/`

[3] Latent Semantic Indexing is an indexing and retrieval method, based on the principle that words that are used in the same contexts tend to have similar meanings.

method the terms were ordered by decreasing score and the top five were considered to build the summary. Thus they discovered that pieces of source code can be automatically summarized using text retrieval techniques.

For what concerns this document, we appreciate the extractive approach, and the choice of using graduate students for the experiment.

2.3 Emails

Emails constitute a composite, chronologically ordered, and structured (due to the many fields they have, like author, subject, date, body, ...) material that can be summarized with many different methods: either with an abstractive approach or an extractive one, considering either a single email or a complete thread, extracting either keywords or sentences, using either determinism or non-determinism.

Carenini et al. [4] proposed a schematic and synthetic solution: creating a "fragment quotation graph" that captures the core of the conversation, by using clue words to measure the importance of each sentence. Their idea was that by exploiting quotations in emails, a good structural representation of the conversation could be provided.

A fragment quotation graph $G = (V, E)$ is a directed graph where each node $u \in V$ is an email in the folder, and an ordered edge (u, v) means that node u replies to node v. They defined the quotation depth of a line as the number of quotation markers ">" in the prefix. Assuming that "*any new fragment is a potential reply to [...] quoted fragments immediately preceding or following it*", an edge is added between such neighboring nodes.
A clue word from a node in this graph is a stemmed term that is contained also in its parent node(s) and/or its child node(s). They also used the concept of hidden email, which is an email quoted by at least one email in the folder but not present. If a fragment is a hidden email, edges are created within the neighboring block.

Once terminated the construction of the graph, they used it to determine the most relevant keywords to be included in the summary. The assumption was that words strictly related to the current topic would occur more frequently in successive replies. Thus they built a tool called ClueWordSummarizer that determined the score of a given word with the statistical formula:

$$ClueScore(CW, F) = \sum_{parent(F)} freq(CW, parent(F)) + \sum_{child(F)} freq(CW, child(F))$$

Afterwards they set up a user study to evaluate the extractive summaries produced by ClueWordSummarizer, managed as follows:

1. They selected 20 email threads from the Enron email dataset[4].
2. They hired 25 undergraduate or graduate human summarizers.
3. Every person reviewed 4 conversations in 1 hour (so each conversation was reviewed by 5 different summarizers).
4. A humanly generated summary had to contain 30% of the original sentences.
5. Every selected sentence had to be classified either as essential or optional.
6. They assigned a score to each sentence to evaluate it according to human selection.

From this study they discovered that, sorting the sentences by the attributed score, 17% of the ones in top-30% sentences belonged to hidden mails. Therefore this kind of emails contains relevant information that should be included in a summary.

In our opinion, the main drawback of this approach is that they exploited only deterministic characteristics proper of the email structure, without considering, for the extractive operation, frequent features found in human generated summaries. Anyway, we consider valid the extractive approach, the choice of using graduate and undergraduate students for the experiment, the idea of limiting the length of the summaries written by annotators, of dividing selected sentences into essential and optional, and using this to assign a score to every sentence.

Lam et al. [7] 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). Such system used heuristics to remove email signatures, fields and quoted text, and to identify names, dates and companies cited in the email by using regular expressions and training data. If an enclosing email thread exists, this algorithm processed the email ancestor to have additional information about the context. Even though choosing to summarize only the

[4] http://www.cs.cmu.edu/~enron/

email itself and its ancestors was aimed at obtaining summaries of limited length with respect to the original size of the thread, they noticed that the length of the summaries increased proportionally to the deepening of position of the last arrived message.

At a later time they set up a user evaluation test:

1. They hired 4 participants.
2. They let them run the system on 10 of their own messages.
3. They asked the users about their experience with the software.

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. In conclusion, probably due to the fact that they deterministically based the system's functionality on the reply structure of the thread, they created a software to prioritize the emails.

In our opinion, the strongest contributions of this work are the attempt to remove redundant sentences in the produced summaries, and to have found a hybrid method between whole thread and single email summarization. We inherit the idea of distributing to the users involved in the test a pre and post questionnaire, to register his/her education and to gather impressions about the evaluation.

Rambow et al. [10] proposed a technique based on a sentences extraction approach, by exploiting machine learning. They created a golden set of human generated summaries as follows:

1. They chose a corpus of 96 email threads treating mostly of events planning.
2. Each thread contained 3.25 messages on average.
3. Each thread was summarized by 2 annotators.
4. The required features of the summaries were the past tense, speech-act verbs and embedded clauses, length of the summary between 5% and 20% of the original text but no longer than 100 lines.

Once terminated the test, they used the outputs to rate the similarity of each sentence in a thread to each sentence in the corresponding summary, excluding quoted phrases. For each sentence in the thread they saved the highest similarity score. Then they decided a threshold: Sentences with score above such a value were marked with "Y" (meaning that they should be included in the final summary), the others with "N". About 26% of the sentences resulted to be marked as "Y".

Afterwards they started the machine learning exploration. 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. The set "basic" included standard features that are usable for all text genres (like the absolute position of the sentence in the thread, the length of the sentence, relative position of the sentence in the thread, if the sentence is a question, etc). Adding to the set "basic" some features obtained considering the thread broken into messages (absolute position of a message in the thread, relative position of a sentence in its message) they had the set "basic+". Adding again other features specific of the structure of the thread (overlap of subject words with sentence words, number of responses to a message, number of recipients of a message, if a sentence follows a quoted portion in the message) they had the set "full". The framework used to perform machine learning and automatically obtain a sentence classifier was Ripper[5]. This tool takes as input a set of feature names with possible values, and training data. The output is a classification model (in the form of a sequence of if-then rules) for predicting whether a sentence should be included in the summary or not. Two of the most relevant and intuitive rules they discovered were that questions at the beginning of a thread that are similar to the entire thread subject should be saved in the summary, and that sentences which contain most of the words written in the thread subject, and which have a higher number of recipients should be saved as well.

After having extracted the relevant sentences, they wrapped those with the senders' names, the delivery dates, and a speech-act verb (implemented as a function of the structure of the email thread).

We appreciate many ideas of this document: the extractive approach, the way they managed the experiment with human annotators and, mostly, the employment of machine learning to understand which features make a sentence worthy to be included in a summary.

We analyzed three cases of emails summarization:

[5]RIPPER is a rule-based learner algorithm that generates a set of rules to identify classes while minimizing the error.

- [4] that used a deterministic "fragment quotation graph" and sentences extraction through clue words.
- [7] that tried an abstractive procedure on entire threads, by passing through sentences extraction on single emails and by exploiting reply chain structure.
- [10] that started with a golden set of human generated summaries, individuated important features, and passed them to a machine learning tool.

We seemed to understand that our choice of using extractive summarization is confirmed to be good, because systems [4] and [10] are more reliable in terms of quality of the product.

2.4 Speeches

A relatively recent field in summarization research is spoken speeches. Researchers made big progresses in summarizing conversations treating an individual topic. Less work has been done for unrestricted conversations.

Murray et al. [8] used a sentences extractive approach to demonstrate that conversational features in a machine learning classification framework produced better results than systems working only on a specific domain. They set up an experiment by performing in parallel email and meeting summarization. They used logistic regression classifiers and two corpora: the Enron for the first type of summarization and AMI[6] for the second one.

For what concerns the emails, they built a golden set as follows:

1. They selected 39 threads.
2. They hired 5 annotators per thread.
3. The participants were asked to choose 30% of the sentences.
4. The participants had to mark each sentence as “essential” or “optional”. The essential ones weighted three times as highly as the optional ones.
5. They trained a binary classifiers.

For what concerns the meetings part:

1. They had 96 meetings in the training set, 24 in the development test, and 20 for the test set.
2. For each meeting they hired 3 annotators.
3. Every annotator had to wrote abstract summaries and extracted transcript dialogue act segments (DAs).
4. To maximize the likelihood that some data was informative, they considered a DA as positive if it was linked to a given human summary, otherwise as negative.

To evaluate email threads summaries, they implemented the pyramid precision scheme [2]. To evaluate meeting summaries, they used the weighted F-measure metric, relying on human produced summaries. The idea is that DAs linked many times by multiple annotators are most relevant.

Talking more specifically about spoken speeches, they implemented a system called ConverSumm. Such a system exploits conversational properties of different types: Length features, structural features, features related to conversation participants, and lexical features. The evaluation results showed that some features are effective in both summarization subjects (like ClueWordScore [4], $Sprob$ which estimates the probability of a participant given a word, $Tprob$ which estimates the probability of a turn given a word, the length of a sentence), while others (such as if the current participant began the conversation, or normalizing the sentence length by the longest sentence in the turn) are more informative for emails.

Thus, using multiple summarization techniques, they stated that the system ConverSumm could be effective in the implementation of a unique system working for many conversational subjects (emails, bug report, meetings...). We inherit also by this paper the choice of limiting the number of sentences the annotators can selected during the experiment, and the requirement of dividing them into “essential" and “optional". We do not consider instead the abstractive human generated summaries.

Murray et al. [9] focused their research on automatic summaries of business meetings. Assuming that human abstractive summaries are to be considered as the best possible performance which automatic approaches aim at,

[6] http://corpus.amiproject.org/

they based their method on statistical formulas. They considered mainly the Maximal Marginal Relevance [5], and the LSA[7]. To investigate how automatic metrics match with human judgements, they set up an experiment with humans:

1. The collection used was 6 meeting recordings from the ICSI Meeting corpus [8].
2. They had 3 or 4 annotators per record.
3. Every participant had at his disposal a graphical user interface to browse a meeting with previous human annotations, transcription time-synchronized with the audio, and a keyword-based topic representation.
4. The test itself was composed by an abstractive and an extractive part. The former required the annotator to write a textual summary of the meeting addressed to someone fairly interested in the argument; the latter required to create an extractive summary.
5. They were asked to select sentences from the textual summary to determine a many-to-many mapping between the original recording and the summary itself.

They also exploited the ROUGE toolkit[9] to systematically evaluate the system since it is based on *n*-gram (sentence fragment composed of *n* words) occurrence between automatically generated summaries and human written ones.

They carried out also a human evaluation test:

1. They hired 5 judges with no familiarity with the ICSI corpus.
2. Every judge evaluated 10 summaries per meeting (totally 60).
3. They had at disposal a human abstract summary and the full detailed transcription with links to the audio.
4. The judges had to read the abstract, consult the full transcript and audio, taking not more than 20 minutes totally.
5. Without consulting again the given material, they had to answer to 12 questions per summary (6 regarding informativeness and 6 regarding readability).

They declared to be quite satisfied with the consistency of human judges; on the contrary they were not so confident about the ROUGE evaluation approach. From the obtained results, it seems that ROUGE is not sufficiently robust to evaluate a summarization system. With this document, we definitively consider the idea of carrying out an experiment with human annotators to build our golden set, and of including in the test material some questionnaire for the judges.

2.5 Extractive summarization

After explaining how extractive methods have been applied to all the discussed fields, we highlight some procedural differences.

In the case of bug reports [11], the extractive method was accompanied by a human test where the required summaries were of abstractive type.

In the summarization of pieces of source code [6], a pure form of extracted snippets was applied. Extractive selection of keywords was required in the human experiment too.

For what concerns emails, we saw different kinds of approach: a sentences extractive method based on the previous withdrawal of clue words [4], an abstractive summary of a whole email thread by applying already existing extractive techniques over the single messages [7], and a sentences extractive approach applied over the entire thread [10].

In the spoken speeches field, we discussed both approaches: sentences extraction integrating parallelism with email summarization correspondent techniques [8], and abstraction [9].

Almost all the researchers who used an extractive general approach have declared to be satisfied of the obtained result. We can not say the same for the ones who chose an abstractive general approach. *Lam et al.* [7] realized, thanks to the user evaluation, that their system was useful only to prioritize the emails but not to eliminate the

[7]Latent Semantic Analysis is a technique used in natural language processing to analyze the relation between a set of documents and the terms they contain.

[8]`http://www.icsi.berkeley.edu/speech/mr/`

[9]`http://www.germane-software.com/software/XML/Rouge`

need for reading them. *Murray et al.* [9], as a result of a user evaluation, realized that their summaries were good according to human judges, but their system was not so reliable according to ROUGE evaluation toolkit. Drawing a conclusion from the analyzed research, we consider extraction for the implementation of our system, and abandon the abstractive option.

2.6 Considered techniques

We consider our approach as a mixture of all the methodological steps we found to be valuable while reading the documents about the topic. The ideas we decide to embrace are the following:

- The employment of human testers to gather useful features to be used in the computations aimed at extracting the sentences from a text (as in the cases [10] and [8]).
- The employment of graduate or undergraduate students in the experiment, because the probability they understand the topics treated in the emails is higher ([6], [4]).
- Limiting the summary length written by human testers, because too long summaries do not solve the problem of reading very long text ([4], [10], [8]).
- The requirement of dividing the selected sentences into “essential" and “optional" and give different score to a sentence according to its category ([4]).
- Human classification of sentences to assign a binary score to every phrase, which will be useful in machine learning phase ([10]).
- Giving the human testers pre and post questionnaires to check if their choices are influenced by some external factor like tiredness ([7], [9]).
- Machine learning ([10], [8]), because we observed that the best working systems are the result of a procedure involving it.

We strive for creating an approach similar to the one proposed by *Rambow et al.* [10]. The differences are that they aimed at summarizing whole threads, while we aim at summarizing single messages, and that we implement also a software system.

3 Methodology

Since our purpose is to implement a system to simulate human choice of important sentences from a text, we need to create a systematic environment to explore how such a spontaneous action happens. A way to do this is trying to derive a function whose parameters are typical features of human summaries.

In other words, having this email:

Subject: [argouml-dev] Merging ArgoEclipse fork of eUML implementation
From: Tom Morris (tfmogmail.com)
Date: Apr 8, 2010 1:23:42 pm
List: org.tigris.argouml.dev

When it was uncertain that ArgoUML would ever adopt the EPL, I created an ArgoEclpse-hosted fork of the eUML implementation for all my work. Since all the code at the time of the fork had been created by just two people, I got Bogdan's written permission to relicense everything and dropped the BSD license entirely.

Now that ArgoUML has adopted the EPL, it makes sense to merge the two code bases back together so that they can each benefit from changes unique to the other.

For modules which still haven't been touched by anyone else, I'm going to use the simplified EPL-only headers. For modules which have had changes made by others, I'll seek written permission from the other authors to relicense as EPL-only and remove the BSD copyright notice. For any modules that I can't get everyone to agree, they'll need to retain both copyright headers.

If you know you're one of the people who has modified the ArgoUML eUML model implementation, you can reply to this thread indicating whether or not you approve of the EPL-only headers. Note that none of this has any effect on the license which is in effect for these modules. It's simply a matter of reducing the amount of text in the headers. In all cases, all earlier versions of the modules are BSD licensed and all later versions are EPL-licensed (even if they contain BSD copyright notices) since the EPL has more restrictive terms than the BSD license.

I'll be doing the first batch of the simplest merges today and working my way through the progressively more complex ones.

Tom

The desired output of our system should be a summary similar to this one:

When it was uncertain that ArgoUML would ever adopt the EPL, I created an ArgoEclpse-hosted fork of the eUML implementation for all my work. Since all the code at the time of the fork had been created by just two people, I got Bogdan's written permission to relicense everything and dropped the BSD license entirely.
Now that ArgoUML has adopted the EPL, it makes sense to merge the two code bases back together so that they can each benefit from changes unique to the other.
I'll be doing the first batch of the simplest merges today and working my way through the progressively more complex ones.

In the following subsections we present our steps: some pilot explorations to establishe which kind of extractive summarization we consider, the detailed execution of the needed human tests, the automated creation of benchmarks, and the machine learning exploration.

3.1 Pilot exploration: keywords vs sentences

The first questions to be answered are:

1. **What do we have to extract from an email?**

2. **Is extracting only keywords more schematic and easy to understand than selecting entire sentences, or is it the contrary?**

3. **Should we propose both extraction methods?**

To give us answers, we performed a **pilot test** with one person (*participant A*):

- 6 email threads chosen from ArgoUML[10] mailing list are the material for the test.
- *Participant A* must sum 3 of these by extracting keywords and the other 3 by extracting sentences.
- We annotate the outputs with the *type of included email messages*, the *length of the thread* measured in pages, the *time used to summarize the thread*, and the *keywords* or *sentences* extracted.

The material is in Appendix A.1.

Here we report the considerations of *participant A* immediately after the test:

- Extraction of sentences is easier and more natural than extraction of keywords.
- Keyword extraction takes almost twice the time.
- In keyword extraction, digrams, trigrams[11] and very small noun phrases often must be selected as a single keyword.
- If a person does not know the system, understanding which method signature or variable or constant should be considered is hard, because everything seems to have the same importance.
- With sentence extraction applied to pieces of source code, only comments are considered.
- By extracting keywords, there is a high risk of missing relations among them.
- Usually chosen sentences are subsets of the chosen keywords, because with the keyword extraction a person is tempted to underline also many concepts that are not so important for the general comprehension of the email.

Some considerations seem counterintuitive. For example, the fact that the participant took more time for keyword extraction than for sentence extraction, and the fact that extracted sentences are a subset of the sentences containing keywords. Other remarks that we want to investigate more deeply are the unavoidable need to extract digrams and trigrams instead of single terms.

We decided to carry out a second pilot test with another person (*participant B*) just to confirm (or to put even more into question) our supposition. The test material and the assignment are the same; the test output can be found in Appendix A.2.

We propose to participant B a questionnaire:

a) Is it easier to extract main keywords or sentences from the emails? *"Keywords."*

b) Is it faster to extract main keywords or sentences from the emails? *"Keywords."*

c) During keyword summarization, did you sometimes feel the need of underline digrams/trigrams?
"Yes, especially in natural language when there are negations."

d) Since you already know what the system the emails are talking about does, did you find difficult to understand the real meaning of the emails?
"No, just in extreme cases like when they talk about the tools used in the implementation."

e) When you extract sentences from the emails containing code/script/compilation report, did you underline pretty much just the comments or also many parts of the code itself?
"No, I also considered classes and methods name."

f) Did you feel in keyword summarization that some logical link among subordinates went lost?
"Yes, especially when to a sentence with positive sense a phrase with negative sense follows (or viceversa)."

[10] `http://argouml.tigris.org/`

[11] An *n*-gram is a contiguous sequence of *n* words belonging to the same sentence. So a digram is formed by two words, a trigram by three words.

g) According to you, is more correct either to say that keywords are a subset of the summary by sentences (i.e. on the same email, all the keywords you would underline are contained in the underlined sentences), or to say that sentences are a subset of the keywords (i.e. on the same email, there exist many underlined keywords that would not be contained in the chosen sentences)? *"The second option.*"

By the end of the pilot exploration, these are our answers to the initial questions:

1. **Our system will extract sentences to form a summary.**
2. **Extracting only keywords is neither more schematic because the risk of selecting too many redundant words is high, nor more intuitive because single keywords have difficulty to well transmit the relations among them.**
3. **Due to lack of time, we prefer to focus only on analyzing the sentence extractive method.**

3.2 A human summaries collection

Since we have a focused idea of the summarization technique we want to implement for our system, we are able to start the procedure to gather a consistent set of material that we use as human golden standard (a sort of highest ideal result we desire to reach as automatic output of our system).

Therefore we start an experiment in the form of a test involving humans, called **"Benchmark creation**". Generally speaking, a benchmark is a point of reference for a measurement. In our case, it looks like a list of features with many values assigned to each one. A way to obtain such values is gathering them from a set of human generated summaries.

We divided this testing phase into two rounds called **"Round 1**" and **"Round 2**". We managed "Round 1" in the following way:

- We hired 9 participants (5 undergraduate USI students, 2 graduate USI students, 2 programming experts).
- We chose as testing mailing list the ones from ArgoUML and FreeNet[12] projects (1 code thread, 3 natural language threads, 1 patch thread, 1 strace thread for each system, with every thread summarized by 3 participants).
- We distributed to every participant support material (Appendix B.1) which included:
  - a READ_ME file with the rules of the test
  - a form to fill in with personal data and his/her IT experience before the test
  - a personalized package of 4 different email threads in pdf format (2 from System A which is ArgoUML and 2 from System B which is FreeNet)
  - brief descriptions of the reference systems taken from Wikipedia
  - a questionnaire to be filled after the test
- We imparted some instructions:
  - Taking no more than 45 minutes per system to push participants to stay focused during the extraction of the sentences and to not take too time, because in this case the work would become annoying.
  - Selecting about 30% of original sentences.
  - Marking them as *essential* or *optional* to make possible to indicate the most informative parts.

The analysis of "Round 1" outputs highlights some recurrent features of the extracted sentences:

- The first sentence of a thread (after the usual "hi"/"hello") is almost always highlighted as essential.
- The first sentence of an email is very often highlighted, unless it is short and contains high percentage of stop-words.
- If a phrase contain a web link or it is constituted just by a link, it is usually underlined (we think this is very useful for a programmer, because maybe that link contains some important information that must be retrieved at some point during the project).

[12] `https://freenetproject.org/`

- If a sentence contains some citations or some quoted phrase, usually it is highlighted.
- The last sentence before a strace is always underlined with 3 “essential".
- For what concerns straces, often just the first line is underlined.
- About the 60% of natural language sentences containing some of the substantives contained in the subject of the thread, are underlined with 2 or 3 “essential".
- About the 15% of code lines containing some of the substantives contained in the subject of the thread, are underlined with 1-2 “essential". Probably this is because code often repeats, so highlighting many similar lines would be useless.
- Many short sentences (7-8 words) containing at least one completely capitalized word are highlighted with 1-2 “essential".
- Greeting sentences (both at the beginning and at the end of an email) are never underlined.
- In natural language blocks, 90% of the sentences containing the name of a class or the name of a method is highlighted with 2-3 “essential".
- If the second-last or last sentence is a question, it is often underlined.

At this point we were ready to set the “Round 2" test and gather more data for filling our benchmark. The division into two rounds of the testing phase was turning out useful to apply some changes:

- We hired again 9 participants (5 undergraduate students and 4 graduate students in Computer Science)
- We gave them the personalized package this time structured as follows:
  - FreeNet is System A and ArgoUML was System B.
  - For each participant we randomized the order of the questions in the post-test questionnaire, hoping to obtain totally un-biased answers.

The output of Round 2 test basically confirmed the recurrent features of the sentences extracted in Round 1.

3.3 Critical analysis of "Round 1" outputs

(A short preamble: the following subsection is a statistical digression on the results of Round 1. It is not essential to the comprehension of the rest of our work. So we let the reader totally free either to skip it or to read it as an in-depth examination.)

We were particularly intrigued by the seen results: We thought that they could be useful to realize whether the structure of Round 2 test had to be improve.
We were interested mostly in understanding which level of agreement there exists among different human summaries of the same type of thread (e.g. a thread containing source code, a thread with only natural language, etc). So we produced an average bar charts of the results gathered for every thread type.

We discovered that participants agree more on threads containing pieces of source code than on other types of threads; this means that the number of sentences selected by one, or two, or three persons is almost the same, with no overwhelming brunt of the number of sentences selected by just a single participant.

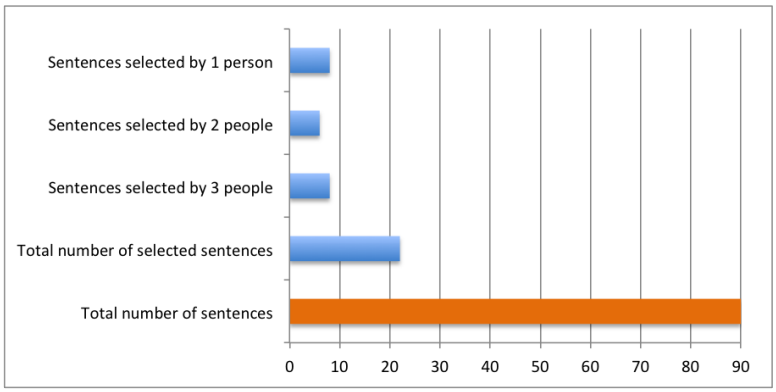

Figure 1. Average test result for CODE thread-type

People seem to disagree more on thread constituted only of natural language.

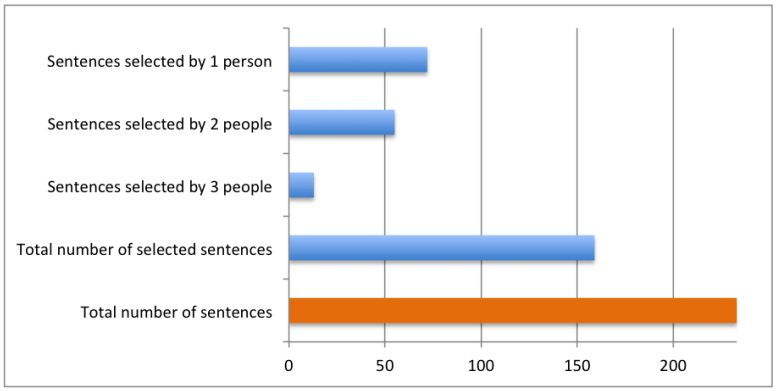

Figure 2. Average test result for NATURAL LANGUAGE thread-type

The average behavior for what concerns patch thread type follows the one of code thread-type.

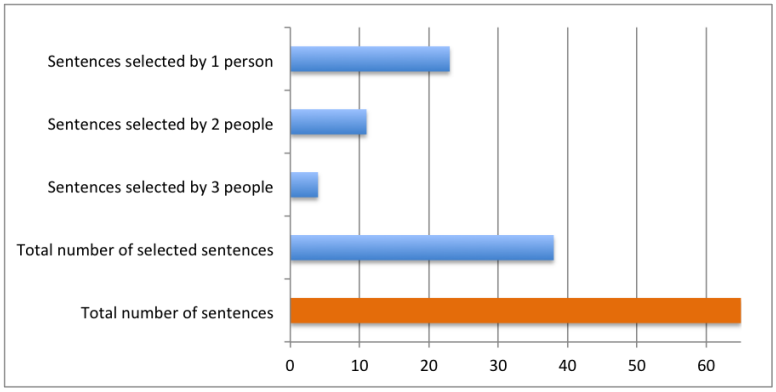

Figure 3. Average test result for PATCH thread-type

While for strace thread type follows the one of natural language thread type.

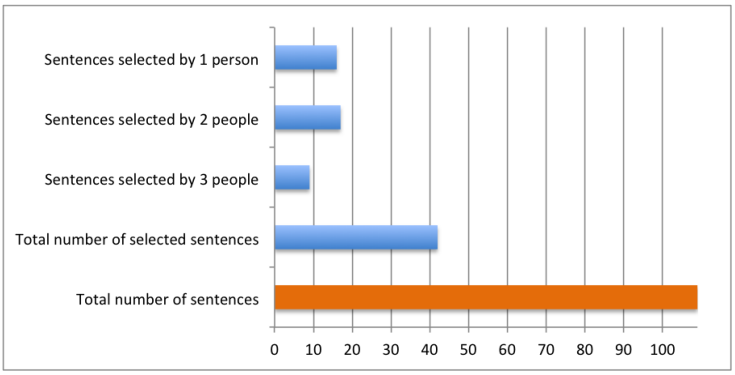

Figure 4. Average test result for STRACE thread-type

Perhaps such a behavior is due to the fact that people tend to underline always the same things in pieces of source code: class names and method signatures. We thought that this happens because they do not represent machine instructions, but only the functionality of the piece of code they contain. On the contrary, in case of natural language, a human subject is able to better interpret the text and everyone can build a personal opinion about the weight of every sentence.

Referring to the post-test questionnaire, the participants had to write for every question an integer vote between 1 and 5, with the corresponding extreme values indicated in the question itself. We discovered that on average people did not find summarizing thread from a system more difficult than from the other, but they found the division into essential and optional sentences harder for System A rather than System B. This is an interesting aspect to be inspected: is this task really harder for ArgoUML, or the results are due to the fact that such system was required to be summarized first and so participants were still unaccustomed to this job?

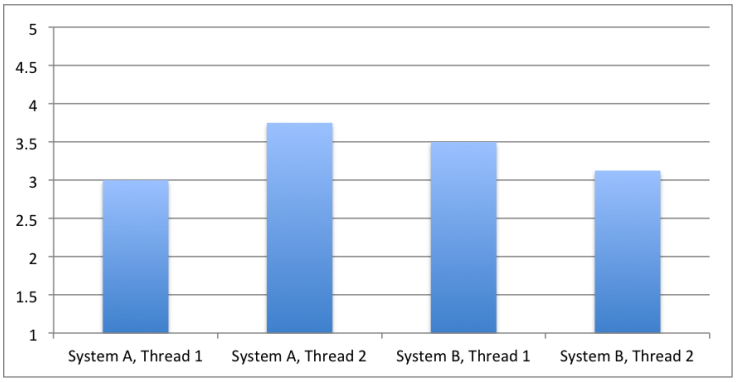

Figure 5. Average on answers to question “How difficult was it to summarize emails by sentences?" [1 = “very easy", 5 = “very hard"]

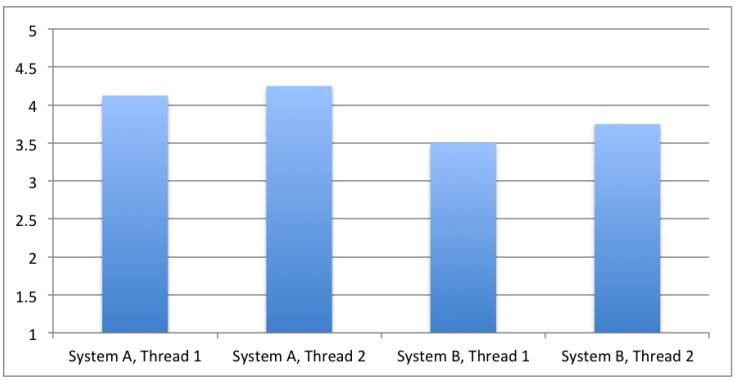

Figure 6. Average on answers to question “How difficult was it to divide sentences into essential and optional?" [1 = “very easy", 5 = “very hard"]

People found a discrete amount of project-specific terminology in every thread, except for the first thread of System B.

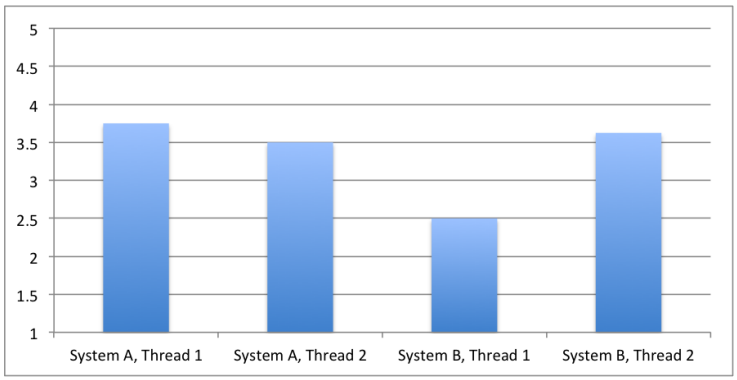

Figure 7. Average on answers to question “How large was the amount of project-specific terminology used in the threads?" [1 = “none used", 5 = “very large"]

Despite this and the fact that they knew very little about the systems, although a little bit more for FreeNet, they had not much trouble in comprehending the meaning of the threads. Therefore we can expect a certain reliability regarding the extracted sentences.

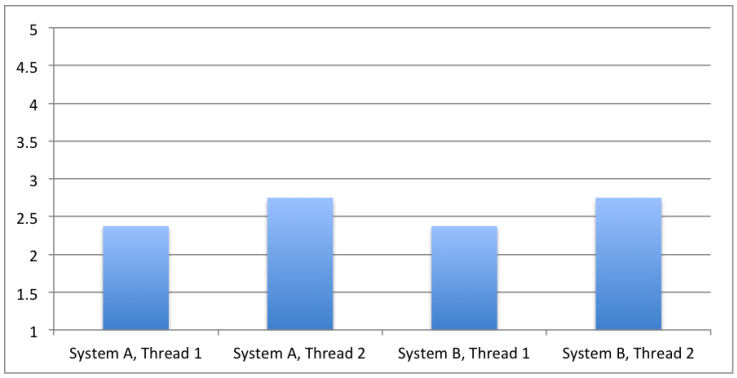

Figure 8. Average on answers to question “How difficult was it to understand the real meaning of the emails?" [1 = “very easy", 5 = “very hard"]

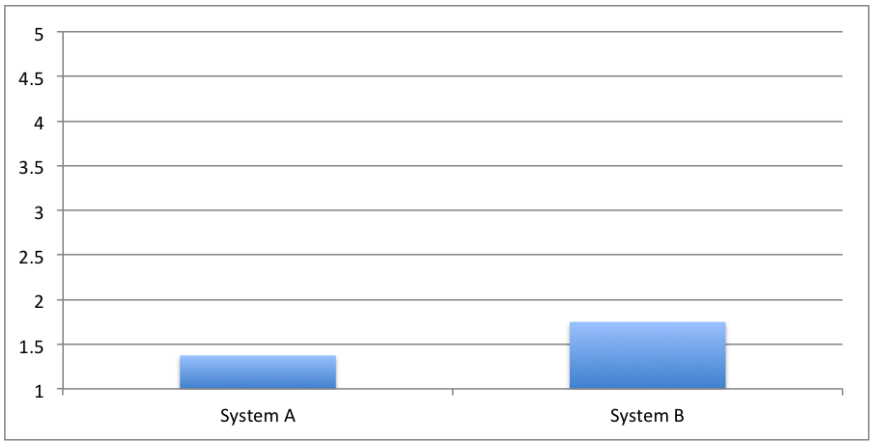

Figure 9. Average on answers to question “How much did you already know about the discussed system?" [1 = “nothing", 5 = “I am expert of it"]

The results of Round 2 showed that on average people took less effort to summarize the threads. Since we switched ArgoUML (now System B) with FreeNet (now System A) and participants again said that dividing sentences of System B into “essential" and “optional" was slightly more difficult, we could conclude that this task is really harder in the case of ArgoUML (in other words, it is not due to training). People employed in Round 2 seemed to know better (although always in a restricted way) ArgoUML; probably this is the reason why they found its description file more useful than FreeNet’s one. Generally, as for Round 1, participants did not find sentence extraction so difficult. The material showing the facts above is in Appendix B.2.

3.4 Benchmark creation

Once collected a sample of 72 human generated summaries during Round 1, we decided the names of the features to be used as attributes in the benchmark.

ID	sentence identifier
ID_AUTHOR	which author wrote the sentence
ABS_POS_NORM	normalized absolute position of the sentence in the thread
CHARS	how many characters the sentence is composed of
FIRST_AUTH	1 if the sentence has been written by the author with $id = 1$, otherwise 0
FIRST_AUTH_ANSW	first author who answered to the sentence
FIRST_NO_GREET	first sentence of the mail which is not a greeting
GREET	1 if the sentence is a greeting, otherwise 0
IS_FIRST	1 if the sentence is the first of the mail, otherwise 0
IS_LAST	1 if the sentence is the last of the mail, otherwise 0
IS_QUESTION	1 if the sentence is a question, otherwise 0
LAST_AUTH_ANSW	last author who answered to the sentence
LINES_ANSWRD	how many lines the sentence answers to
LINK	1 if the sentence contains a web link, otherwise 0
NUM_ADJECTS_NORM	normalized number of adjectives the sentence contains
NUM_ANSWERS	how many answers the sentence received
NUM_CONSNTS_NORM	normalized number of consonants the sentence contains
NUM_NOUNS_NORM	normalized number of nouns the sentence contains
NUM_NUMS_NORM	normalized number of numbers the sentence contains
NUM_STOPW_NORM	normalized number of stop-words the sentence contains
NUM_VERBS_NORM	normalized number of verbs the sentence contains
NUM_VOCALS_NORM	normalized number of vocals the sentence contains
NUM_WORDS	how many words the sentence is composed of
QUOTS	1 if the sentence contains a citation or a quoted sentence, otherwise 0
REL_POS_NORM	normalized relative position of the sentence in the mail
SHORT_CAPT	1 if the sentence is short (less than 8 words) and contains at least one capitalized word, otherwise 0
SUBJ_WORDS	normalized number of words of the thread subject the sentence contains

We added to this set the following features that would have been useful to classify the sentences during the machine learning phase.

HMN_SCORE	score gained from the test
BINARY_SCORE	1 if the HMN_SCORE is different from 0, otherwise 0
CLASS	"non-relevant" if BINARY_SCORE is 0, otherwise "relevant"

For every thread, we manually inserteded all the original text in a CSV file (one sentence per line) and we annotate, for each participant that summarized the thread, a score of 2 points for the sentence if it has been marked as “essential", of 1 point if it has been marked as “optional", of 0 if it has not been extracted at all. Then we extract the value of each attribute from every sentence and we store everything in the same CSV file. When we completed this job, we have our benchmark ready and its aspect is similar to this (the complete view of all the attributes is in Appendix B.3):

I	J	K	L	M	N	O	P	Q
abs_pos_nor	chars	first_auth	first_auth_ar	first_no_gre	is_first	is_last	is_question	greet
0.01351351	12	1	2	0	1	0	0	1
0.02702703	87	1	2	1	0	0	0	0
0.04054054	104	1	2	0	0	0	0	0
0.05405405	61	1	2	0	0	0	0	0
0.06756757	27	1	2	0	0	0	0	0
0.08108108	88	1	2	0	0	0	0	0
0.0945946	106	1	2	0	0	0	0	0
0.10810811	43	1	2	0	0	0	1	0
0.12162162	7	1	2	0	0	0	0	0
0.13513514	5	1	2	0	0	1	0	0
0.14864865	31	0	1	1	1	0	0	0
0.16216216	107	0	1	0	0	0	0	0
0.17567568	67	0	1	0	0	0	0	0
0.18918919	114	0	1	0	0	0	0	0
0.2027027	28	0	1	0	0	0	0	0
0.21621622	7	0	1	0	0	0	0	1
0.22972973	4	0	1	0	0	1	0	0
0.24324324	6	1	2	0	1	0	0	1
0.25675676	66	1	2	0	0	0	0	0

Figure 10. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns I-Q)

Since we had to extract the values for all the features from every sentence of every analyzed email thread and write them in the prepared CSV files, this procedure would have been absurdly long to be done manually. So we implemented a Python program called "**CsvHandler**" to automate the filling.

We largely used the `nltk` library[13] which is great at tokenizing periods, tagging words with the grammatical part of speech they represent, managing stop-words. We used mostly the function `nltk.regexp_tokenize(<string>, <regex>)` to split a sentence into words. It takes a string and tokenizes it applying the given regular expression. We employed the following one:

```
PATTERN = r'''(?x)
   \w+\://\S+
   | i\.e\.
   | e\.g\.
   | \w+(/\w+)+
   | \w+(\\\w+)+
   | \w+((-|_|\.)(?!(as|java|php|c|h)\W|\w+\()\w+)+
   | \w+
   | -\d+
'''
```

To tag the adjectives, the nouns, the verbs in a sentence we used the function `nltk.pos_tag(<words>)`, which takes an array of singular words, and returns a list of pairs `(words, grammatical part of speech)` (e.g. 'VB' for verbs, 'NN' for nouns, ...). To recognize stop-words we exploited the `nltk.corpus stopwords`.

Once run this program on all the email threads, we merged all the results in a single CSV file that we used as input benchmark for the machine learning.

3.5 Machine Learning

There exist many precise and technical definitions for **machine learning**. In simple terms, it is a branch of artificial intelligence aimed to design special algorithms that allow computers to evolve behaviors, by exploiting empirical data. In our research, by giving our benchmark as input to one of these algorithms, we should be able to definitely elaborate which features are most important in human summarization and use that result to automatically build summaries with our system.

We performed this job with the support of the platform *Weka Explorer* [14]. We opened as input file our CSV containing the merged information from ArgoUML and FreeNet, and we removed redundant features (such as the

[13] `http://nltk.org/`
[14] `http://www.cs.waikato.ac.nz/ml/weka/`

singular vote of each participant, that is already included in the sum of them stored at column "hmn_score") that could foul up the results. We started the environment with the following settings:

- We chose as attribute evaluator ClassifierSubsetEval [15] that uses a classifier to estimate the "merit" of a set of attributes.
- The search method was BestFirst *"which explores a graph by expanding the most promising node chosen according to a specified rule*"[16].
- We set "class" as feature to evaluate.

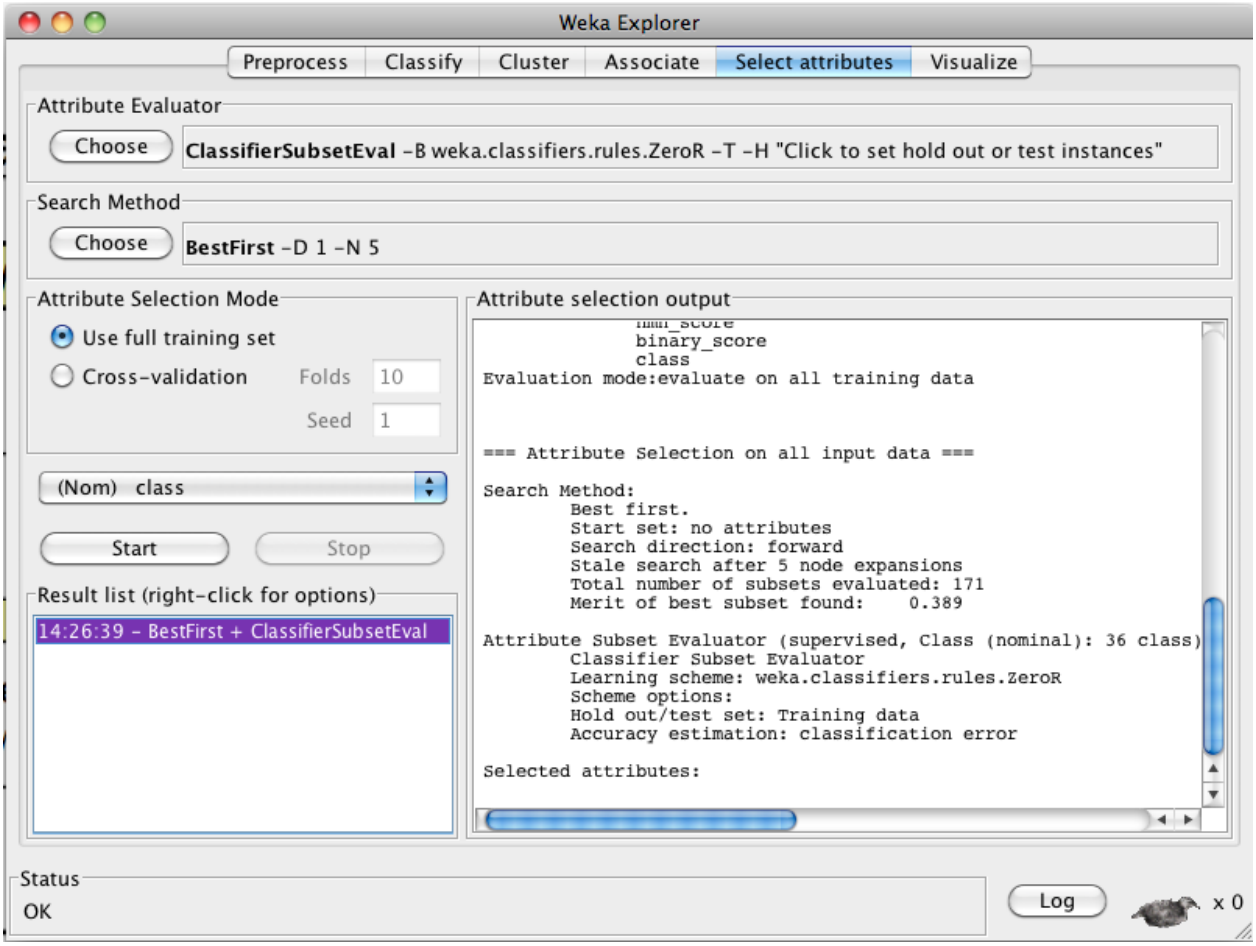

Figure 11. Weka processing platform

Weka Explorer rewarded us with a relevance tree [Figure 12] stating that the 6 attributes which determine the relevance of a sentence are **chars**, **num_nouns_norm**, **num_stopw_norm**, **num_verbs_norm**, **rel_pos_norm**, and **subj_words_norm**. Thanks to the tree we could determine whether a sentence should be included in the summary or not, simply by considering its values of these 6 attributes and going through the conditions written in the tree (starting from the root). If we arrived to a leaf labeled as "relevant", we include the sentence in the summary, otherwise not.

[15] http://bio.informatics.indiana.edu/ml_docs/weka/weka.attributeSelection.ClassifierSubsetEval.html
[16] http://en.wikipedia.org/wiki/Best-first_search

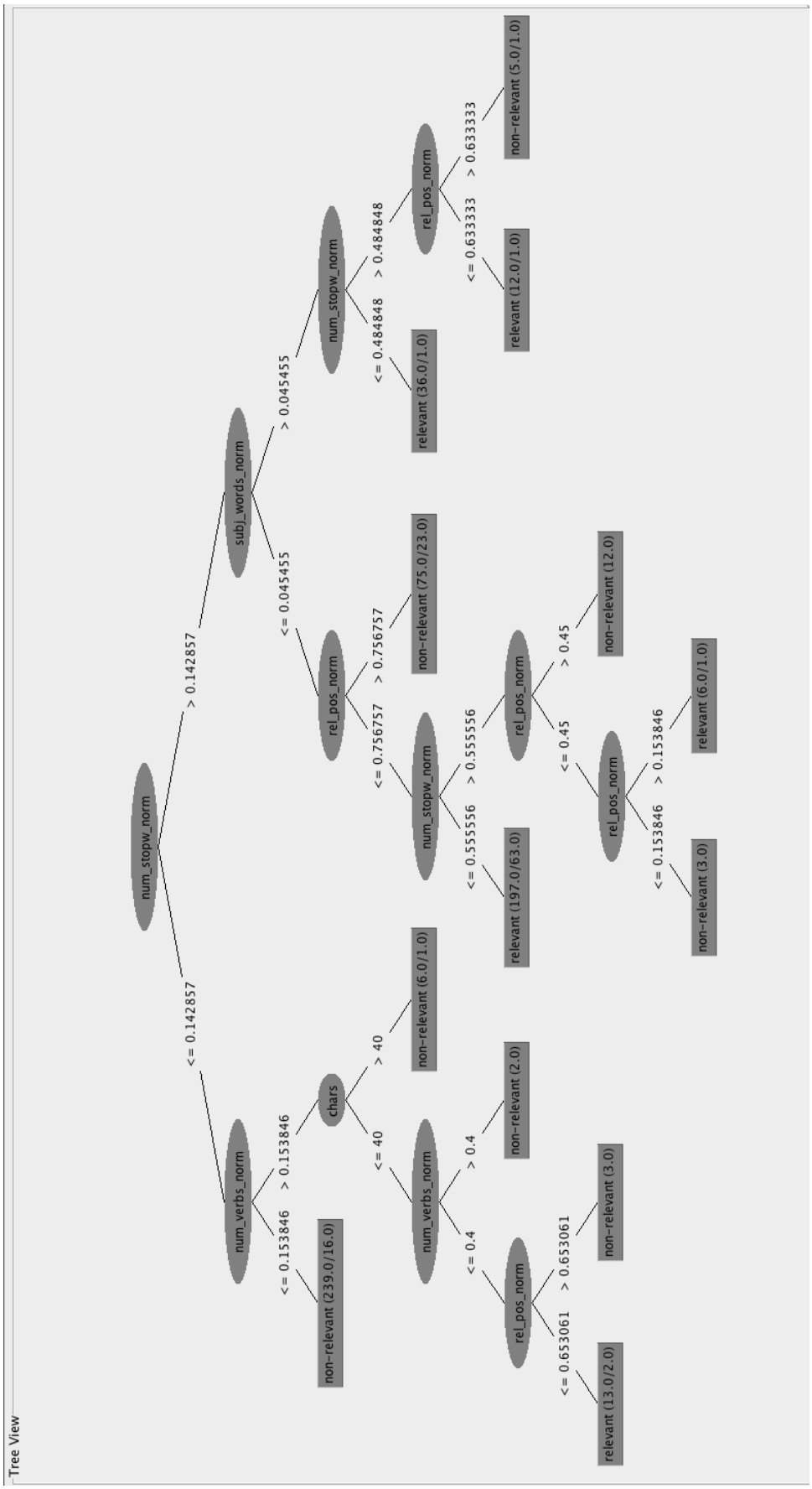

Figure 12. Weka output tree

4 Implementation

4.1 Summit

While the “Part I" tests were in progress, we started to implement a simple summarizer we called “SummIt", just to get familiar with `mbox` file format. We use Python as programming language, the `nltk` library, and the package `mailbox` [17] to handle `mbox` files.
SummIt took as input an email thread, cleaned it from symbols like “=0A" (which is line feed), and removed all the lines starting with “>” because they are replying messages (we do not need repeated sentences).
The produced summary was based on simple word frequency: for each non-stop-word term we computed how many times it recurs in the thread and divided it by the total number of non-stop-words. Then we let the user pass to the program a threshold to determine how accurate the extractive summary must be: for example, if a user passed 3 as parameter, then SummIt returned a summary where extracted sentences contained at least one word that appeared in the thread with

$$\text{frequency} \geq \frac{\text{frequencies\_sum} \cdot 3}{100}$$

Note that the higher the threshold, the shorter the summary.

This program could be run with the terminal. The parameterized command line to use is:

```
$ python SummIt.py -m file.mbox -p 3 -o file.txt
```

where the parameter specified with “-m" is the `mbox` file containing the thread, the one specified with “-p" the desired threshold, and the one with “-o" the output file.

What SummIt showed to the user was this:

```
administrators-macbook-pro-10:model mastrode$ python SummIt.py -m prova.mbox -p 3 -o provo
PEOPLE INVOLVED IN THE CONVERSATION:
Sam Mizanin <sammyuglykidpig@yahoo.com>
"ftpserver-users@mina.apache.org" <ftpserver-users@mina.apache.org>
sebb <sebbaz@gmail.com>
ftpserver-users@mina.apache.org
"Sachin Shetty" <sshetty@egnyte.com>
David Latorre <dvlato@gmail.com>



SUMMARY:

These are
small changes like FtpReply not having isPositive and LocalizedFtpReply con=
structor being Private.I would like to see a detailed release note for eve=
ry version, even smaller changes like the ones that i mentioned.Cheer=
s,
Sam
On 2 December 2011 15:38, Sam Mizanin <sammyuglykidpig@yahoo.com> wrote:
 changes that are done from the previous versions.These are small changes
like FtpReply not having isPositive and LocalizedFtpReply constructor being=
 Private.I would like to see a detailed release note for every version, ev=
en smaller changes like the ones that i mentioned.From nobody Sun May 20 20:48:44 2012
Content-Type: text/plain;
        charset="us-ascii"
Content-Transfer-Encoding: 7bit
We have configured our ftp servers to set idle timeout to 15 minutes.Thanks
Sachin
From nobody Sun May 20 20:48:44 2012
Content-Type: text/plain; charset=ISO-8859-1
Hello Sachin,
It seems this is a bug we should fix.2011/12/10 Sachin Shetty <sshetty@egnyte.com>
```

Figure 13. A sample summary produced by SummIt

However, we essentially implemented this program as a sort of “funny, training coding". Moreover it based the relevance of sentences on a formula without any scientific foundation.
So, we abandoned the project SummIt now on.

[17] `http://docs.python.org/library/mailbox.html`

4.2 An Eclipse Plug-in

Differently from the time of SummIt, now we had an algorithm which was the result of our systematic case study. So it was the moment to officially choose how to implement the system. We had basically two options: either to implement it as an IDE plug-in, or as a separated application.

The second idea was a little bit complex and longer because:

- We should implement an application from scratch and this usually takes more time.
- We should integrate a database management system to keep the entire mailing list close to the project.
- There would be always a link missing: since the operation of manually finding emails related to a specific file is severely time consuming, we would need a sort of tool able to do it quickly.

Therefore we took into consideration the other option: building an IDE plug-in. An IDE (integrated development environment) is a software thought to help programmers in their job, usually composed at least of an editor, a compiler, an interpreter and a debugger. By choosing this solution, we had the summarizer close to the source code and we could easily access emails without having to switch between two applications (the IDE and the summarizer).

There exist many multi-language IDEs nowadays, like NetBeans, Eclipse, Visual Studio. We preferred Eclipse because:

- It is open source (so it is free and modifiable without problems).
- It is a platform completely based on the use of plug-ins[18].
- Eclipse can even be defined as a “set of plug-ins", thus the available documentation is vast.
- There exists also Eclipse Indigo which is a special version of the IDE already set up to run the implementation of a plug-in.

Therefore, we definitely agreed for the option: An Eclipse plug-in.

4.3 REmail Summary

Our base: REmail
Since there exists already the plug-in REmail[19] that allows to visualize all the emails and email threads related to a class (stored in *CouchDB* [20]) just by clicking on the class itself, we integrate our system in it.

Some extensions of REmail are:

- REmail Content which shows in a IDE integrated-browser the content of the selected email.
- REmail Writer which allows to write an email directly from Eclipse.
- REmail Visualization which allows to visualize bar or line charts regarding the emails of the class.
- REmail Rating which allows to rate single emails.

Architecture
We implemented another extension called **REmail Summary**. An important decision we had to take concerned the fact that, using REmail, producing summaries of single emails made much more sense. This was due to the fact that, when a user clicks on an email, that email is passed as input to all the open plug-in extension. So taking the entire thread the email belongs to would be very laborious.

As a preliminary step, we created in package `org.eclipse.remail.views` a class **MailSummaryView** which extended the class ViewPart and that we used to display our summary in a text-area. An instance of this class was defined and set in the general view class **MailView**.
Then we created a package `org.eclipse.remail.summary` entirely dedicated to our summarization technique. Here we implemented the classes **Summary** and **FeaturesExtractor**.

Class Summary stores:

- the input selected email

[18] A plug-in is a non-autonomous program which interacts with another one to extend its functionalities
[19] `http://remail.inf.usi.ch/`
[20] `http://couchdb.apache.org/`

- the produced summary
- the short version of the summary
- the length of the summary
- a FeaturesExtractor instance
- a global tagger

Class Summary behavior:

- `summUpMail()` appends to the same text all the email sentences marked as relevant, and passes the obtained summary to MailSummaryView
- `makeShortSummary()` creates a summary considering only sentences whose relevance leaf is at higher levels of the tree, until reaching 60% of the length of the original message.

The task of class FeaturesExtractor, in a nutshell, is to implement the algorithm outputted by *Weka Explorer*. FeaturesExtractor took as input a Mail object and a tagger, removed the lines of the format *"On <date><time>, <author> wrote:*" right before a replying-email and the replying-lines starting with ">", and stored:

- the subject of the email
- the passed tagger which will be used to individuate the several grammatical parts of speech
- a sentence-id hash map
- a matrix where, for each sentence of the email, all the values of the extracted relevant features are inserted

Dealing with parts of speech
As we saw at Section 3.5, among the important features for the determination of the relevance of a sentence there is also num_verbs_norm. To understand how many verbs there are in a sentence, we used the function `tagString(` $< string >$`)` of the `MaxentTagger` we stored. This tool is a part-of-speech tagger suitable for Java programming, developed by *Stanford natural Language Processing Group* [21]. We installed it in Eclipse very easily by importing `stanford-postagger.jar` in the Referenced Libraries, and by creating a folder `tagger` where to place the English vocabulary files.

Implementation of the relevance tree
The function `determineRelevance()` computes through a long $if - chain$ the relevance of a sentence by accessing the information stored in the features matrix. The result (1 if the sentence is calculated to be "relevant", 0 otherwise) is then stored back in the last column of the matrix itself.

How to
Using such a plug-in is really intuitive. The user:

1. opens, in Eclipse editor, REmail and REmail Summary views
2. lets CouchDB start (if a shared server among developers does not already exist)
3. clicks on some class to visualize the related emails in the REmail box view
4. reads the extractive summary in REmail Summary box view.
5. If the user judges that the produced summary is still too long, we provide a "Shorten" button that returns a skimmed summary which is long at most 60% of the original text. We remind, however, that this is a completely deterministic process, surely not aimed at preserving the human generated-likeness. This functionality is active only for emails longer than 4 sentences.

[21] `http://nlp.stanford.edu/software/tagger.shtml`

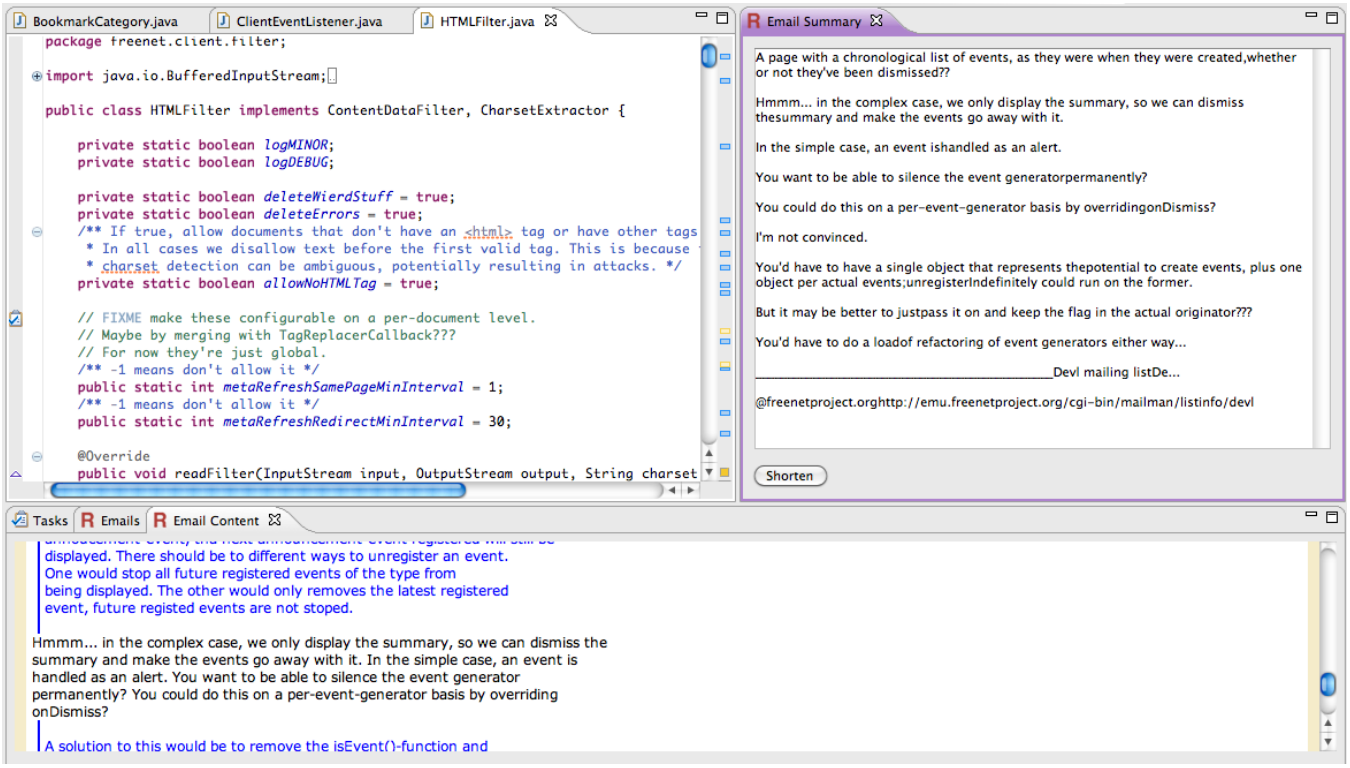

Figure 14. A selected email (taken from FreeNet project) has its content displayed on REmail Content view and the summary in Remail Summary view.

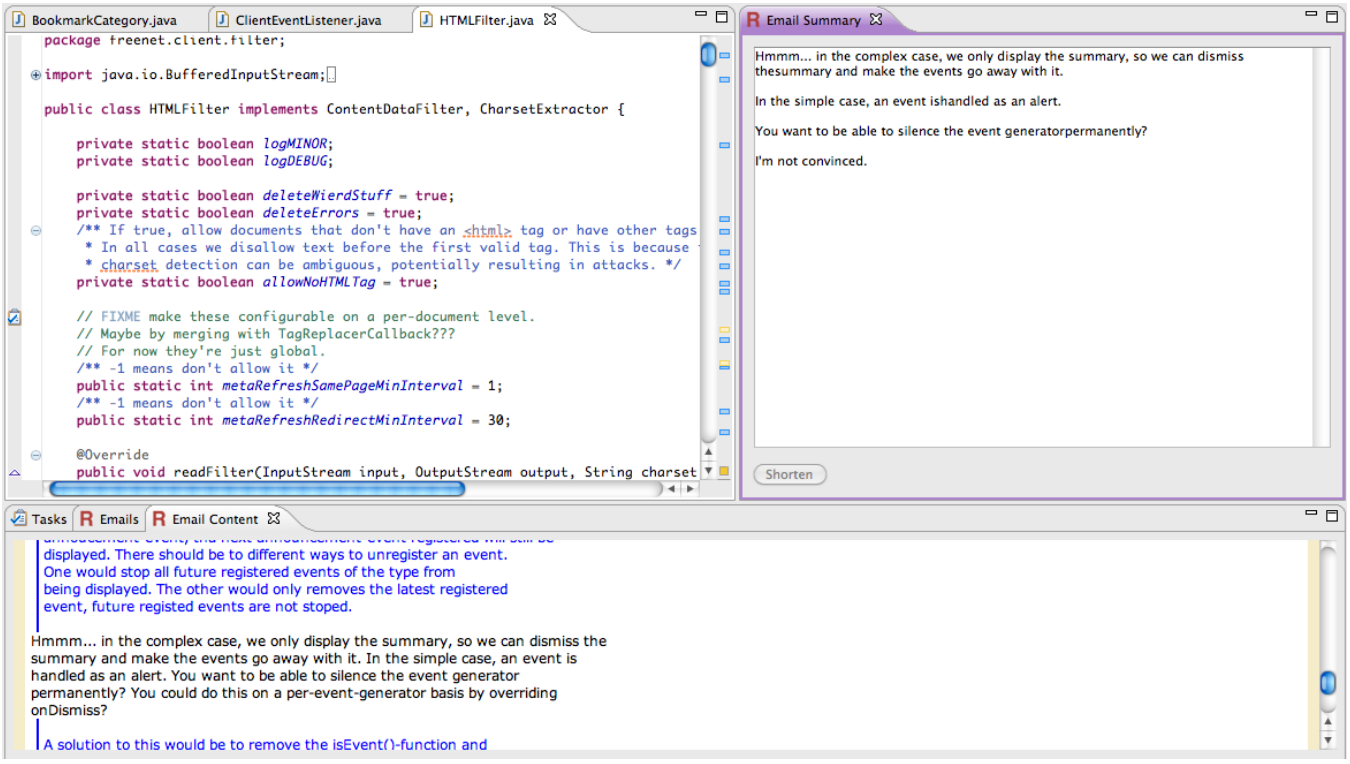

Figure 15. The clicked "Shorten" button generates a summary long at most 60% of the original email (which is the same of Figure 14).

5 Conclusions and Future Work

Our research experience led us to explore the interesting field of human hand-written document emulation. Thanks to machine learning algorithms and training data gathered from tests involving computer scientists, we discovered that there exists a strong correlation between the structural features of a sentence and the choice of humans in sentence extraction.

Taking advantage of this preliminary work, we managed to implement “REmail Summary", an extension to the Eclipse plug-in “REmail”. The benefit of this tool is that it allows to work on the code and quickly access the project mailing list, without having to switch among different applications (in this case the IDE, the database containing the emails, and the software producing summaries).

We observed that for emails containing reply chains REmail Summary tends to select most of the sentences. A real example is this one:

Subject: Re: [freenet-dev] New filter code: stuff that must...
From: Matthew Toseland
Date: 11.06. 2010 07:25

On Friday 11 June 2010 14:43:09 Matthew Toseland wrote:
>BEFORE RELEASING 1251, MUST FIX:
>- Internal error on persistent download for mp3 with the check filter flag enabled. This should be an unfilterable content error.
>- WoT
>- Freetalk
>- Spider
>- XMLSpider
>- FlogHelper
>- BMP filter failure in unit tests.
>- dangerousInlines not dangerousInline
>- Are self-closing tags an issue on truncated filtering in HTMLFilter? I doubt it but double-check what the behaviour is on EOFing when in the middle of a string.
>- ContentFilter charset detection code MUST use readFully, or a loop achieving similar goals. read() doesn’t read the full buffer.
>
>Minor stuff:
>[01:31:51] third, CharsetExtractor taking a byte[] of unspecified size is a bit strange, you should have CharsetExtractor tell the caller how big the buffer should be >
>3fcc1ac770ad0e360b49984521e1977353d933ab
>- close the streams before freeing the buckets
>
>6fd6f8c50ffc4823c2d3c701e0c85c6155039e2a
>- BMPFilter, CSSReadFilter, GIFFilter, HTMLFilter: don’t flush in finally
>- JPEG filter: don’t flush on writing a frame, flush at the end (when returning/exiting the loop).
>
>15b6283e017a26dec96f35820028b84992b5260f
>- should pass the overridden mime type along in places where we generate expected mime type events as well. non-critical.
>- should maybe also pass it in other places where we set the expected mime type on FetchException’s. again non-critical.
>
>9c05c0b1d1840826d6e823dc7f9f46682a3afac8
>(f0dddb70e787388f3a674e9530e44962ce43d1c3 improves a bit)
>- what about just calling getMessage() on FetchException?
>
>I fixed a critical bug in 003aeb0, we were filtering and then discarding the
filtered data and sending the unfiltered data through. Ooops! :) I wasn’t trying to do spencer’s work for him, was chasing something else. Please rebase.

Devl mailing list

De...@freenetproject.org http://freenetproject.org/cgi-bin/mailman/listinfo/devl

This is the produced summary.

filtered data and sending the unfiltered data through. I wasn't tryingto do spencer's work for him, was chasing something else. Please rebase.

And if we click on "Shorten", we obtain this:

filtered data and sending the unfiltered data through.

For what concerns emails containing pieces of source code or stack traces, the result is worse because we focused our research more on dealing with natural language. For example, in the case below the summary produced by our application is empty:

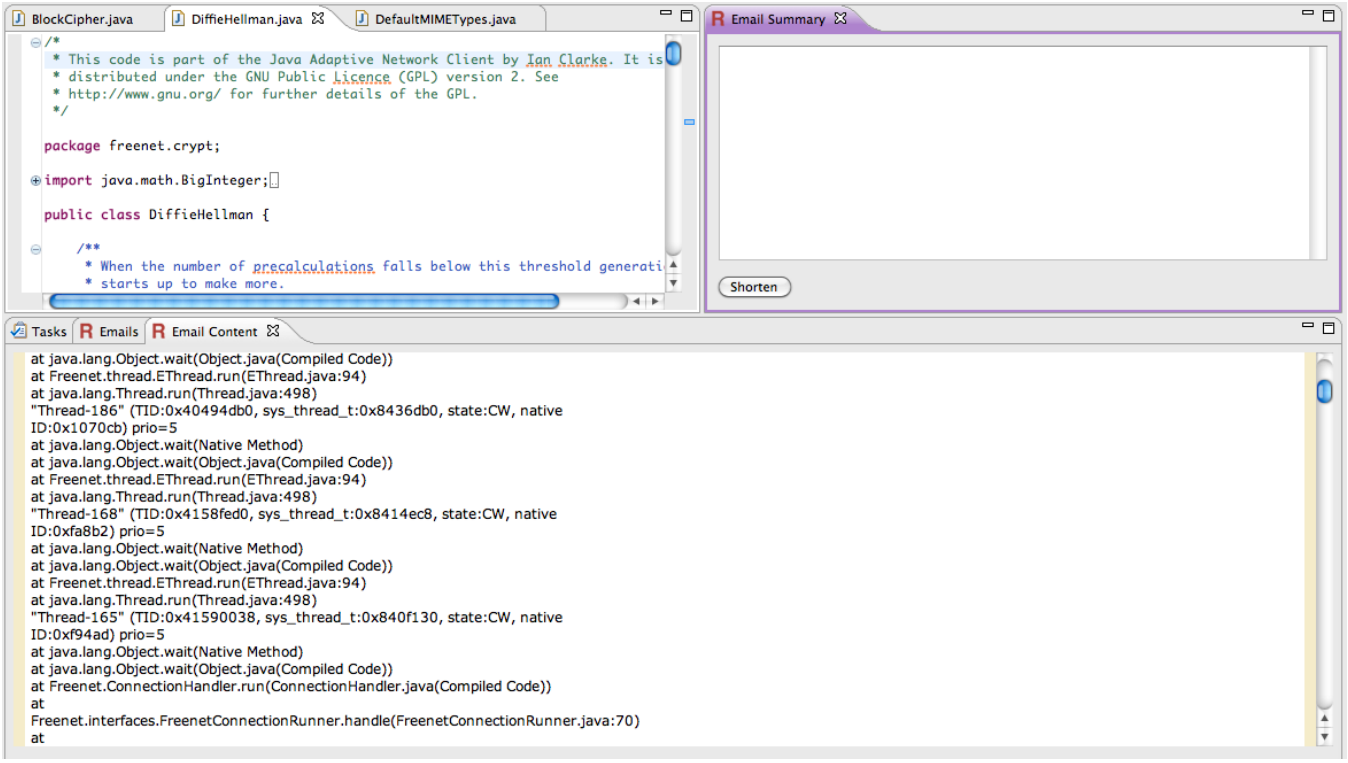

Figure 16. A case in which REmail Summary does not produce a summary for an email containing only a stack trace.

The summary obtained with the rough technique of tree pruning, instead, gives a surprisingly good product. Sometimes one, the right one, sentence has the power to communicate more than tens of lines that discourage the reader. Again, even if the selected sentences are not so conceptually linked, knowing a little bit about the context of the conversation is enough to understand the entire conversation. Analyzing this aspect with a test with human participants would be interesting.

Future work

There is still much work to do to optimize the output of our system. We should find a way to handle pieces of source code, patches, and stack straces in a more suitable way, because for now they are not taken specifically into consideration for automatic summary composition. Moreover we should investigate a clever way to curtail too long summaries, always respecting the purpose of human choice emulation.

Finally we would like to repeat the entire experiment by using a larger set of human annotators. This would allow to discover whether the conditions given by the machine learning would produce summaries that better simulate the human generated ones.

A Pilot exploration material

A.1 First pilot test output

1.

Email id	hup5rxa7dy5rmcwi
Email link	`http://markmail.org/message/hup5rxa7dy5rmcwi`
Email type	Natural language with only few lines of system settings
Email length	2 pages
Extraction type	keywords
Time to summarize	8 minutes
Extracted keywords	ARGO, parameters, different models, user, activity, class, interaction, diagram, augment, embedded functionality, config.xml, GUI, difficult, link, helpful, stereotypes, different languages, modeling environments, UML profiles, config files, default stereotype, types, model, copy, org/argouml/default.xmi, argouml, ProjectProperties, ApplicationProperties, pane, Curt Arnold's xml config file, ATM, database modelling, business modelling, tags, model import functions, XSD, XMI, OMG, XML Schema RFP, questions, ground, dangling roots, standard disclaimers

2.

Email id	lmydo7oxe6rdrhly
Email link	`http://markmail.org/message/lmydo7oxe6rdrhly`
Email type	Natural language
Email length	5 pages and half
Extraction type	keywords
Time to summarize	22 minutes
Extracted keywords	deleted Poseidon, far ahead, stability, Argo, reliability, entice, marketing, discussion, target, users, product, free, stable, full featured UML, developers, bug fixes, enhancements, users/developers need, complex, not enough documentation, Eclipse, Netbeans, Viewlets, direct links, Issuezilla, more users, more developers, suggestions, remove comments, newcomers, irritated, Project members, easy, good documentation, active part, software developers, java code, UML, IDEs, functionality, plugins, module, community, gain, integration, real time demonstrations, XSD, clues, power of Argo, user mistakes, descriptions, not helpful, names, modeling, simple, central zargo

3.

Email id	6cftrd4zi437omrv
Email link	`http://markmail.org/message/6cftrd4zi437omrv`
Email type	Natural language
Email length	1 page and half
Extraction type	keywords
Time to summarize	9 minutes
Extracted keywords	problem, easy to reproduce, RE argouml, modifications, diagrams, CVS, use case, activity, super machine, files, large, out of memory, hardcore, usage, structural representation, behavioral diagrams, structural changes, small explanations, fixed, verify, close, issue #410, not reproducible, Project.java, ProjectMember.java, fix, issue 455, change ArgoDiagram, Andreas, implemented, bells, whistles, bug, old code, findMemberByName, without/with prepended project base name, hack, new method ProjectMember.getPlainName, findMemberByName, file extension

4.

Email id	ajchg4sjbg7vlnhx
Email link	`http://markmail.org/message/ajchg4sjbg7vlnhx`
Email type	Natural language
Email length	2 lines
Extraction type	sentences
Time to summarize	1 minute
Extracted sentences	I start the work for 0.20.alpha5 now. Please no commits.

5.

Email id	huuout2rs74hhzbc
Email link	`http://markmail.org/message/huuout2rs74hhzbc`
Email type	Natural language with an important part of Java code
Email length	4 pages
Extraction type	sentences
Time to summarize	8 minutes
Extracted sentences	Patch for ordering language names alphabetically in “generate all classes" window. `The dialog that starts the generation of classes. Logger. Used to select next language column in case the “Select All" button is pressed. Constructor param nodes The UML elements, typically classifiers, to generate.` I find not safe writing our own sorting algorithm. What about making Language implement IComparable, and call Collections.sort? I find this a good exercise to get started with our code review practices and coding standards. I can’t easily see in the patch where the issue is. The approach sounds reasonable. Formatting in Eclipse sometimes happens by accident. Regarding sorting in algo, add a TODO to the code. Find a nicer way to improve his (of the interested developer) contribution.

6.

Email id	etnen46n3k7i2fnl
Email link	`http://markmail.org/message/etnen46n3k7i2fnl`
Email type	Entirely JUnit tests apart from very few lines of natural language
Email length	6 pages
Extraction type	sentences
Time to summarize	5 minutes

Extracted sentences	Report from Hudson server.
	`Changes:`
	`UML2: adding method field to operation property panel.`
	`UML2: extending model API with Artifact and Manifestation related methods.`
	`UML2: “dummy" implementation for Artifact and Manifestation related methods (not to be used in UML1)`
	`Preparing the build directories in argouml-app`
	`Compiling the sources in argouml-app`
	`Compiling the sources in argouml-core-diagrams-sequence2`
	`Warning: unmappable character for encoding ASCII`
	`Some input files use unchecked or unsafe operations`
	`Recompile with -Xlint:unchecked for details`
	`20 warnings`
	`Total time: 83 minutes 40 seconds`

A.2 Second pilot test output

1.

Email id	hup5rxa7dy5rmcwi
Email link	`http://markmail.org/message/hup5rxa7dy5rmcwi`
Email type	Natural language with only few lines of system settings
Email length	2 pages
Extraction type	keywords
Time to summarize	5 minutes
Extracted keywords	helping, model, interaction, diagram, augment, functionality, make it easier, config.xml, GUI, environments, new default, editing, configure, GUI, application defaults, project specific settings, XML, database, business, collaborate, classes are involved, identify, proposed schema

2.

Email id	lmydo7oxe6rdrhly
Email link	`http://markmail.org/message/lmydo7oxe6rdrhly`
Email type	Natural language
Email length	5 pages and half
Extraction type	keywords
Time to summarize	12 minutes
Extracted keywords	Tutorials, handle, specify, stability, reliability, stability, issue, going for, contribute, discussion, less stable, not stable, full featured, needs, developers, developer, contribute, bug fixes, enhancements, need, problem, perspective, developers, understand, development, contribute, projects, more care, process, connecting, issue, interesting, target, more developers, remove, more stable, more functions, don't agree, approach, solve problem, tools, documentation, test and modify, understanding, development processes, UML, better documented, descriptions, stabilize, add functionality, building plugins, modules, plugins, integration, accomplish, code generation, XML, code generation, code, generation, XMI, XSD, demonstrations, inherent, complexity, mistakes, communicate, modeling, evolving, update, CVS

3.

Email id	6cftrd4zi437omrv
Email link	`http://markmail.org/message/6cftrd4zi437omrv`
Email type	Natural language
Email length	1 page and half
Extraction type	keywords
Time to summarize	8 minutes
Extracted keywords	problem, fairly easy, reproduce, modifications, diagrams, argouml.zargo, argouml, src_new, CVS, has changed, UML diagrams, problems, gather, old project, creating diagrams, use cases, preserved, extremely large, out of memory, structural presentation, behavioral, diagrams, changes, documents, structural diagrams, class diagrams, code improves, won't fix, issue, resolved, fixed, not reproducible, throw it away, keep it, Project.java, ProjectMember.java, problems, Project.findMemberByName, simple, ArgoDiagram, reconsidered, implemented, bug, ProjectMember.getName(), updateProjectName, uncovered, hack, ProjectMember.getPlainName, org.argouml.kernel

4.

Email id	ajchg4sjbg7vlnhx
Email link	`http://markmail.org/message/ajchg4sjbg7vlnhx`
Email type	Natural language
Email length	2 lines
Extraction type	sentences
Time to summarize	48 seconds
Extracted sentences	I will start the release work for 0.20.alpha5 Please no commits

5.

Email id	huuout2rs74hhzbc
Email link	`http://markmail.org/message/huuout2rs74hhzbc`
Email type	Natural language with an important part of Java code
Email length	4 pages
Extraction type	sentences
Time to summarize	8 minutes

Extracted sentences	add a TODO to the code, there's no violation in doing so.
	no need to bring such kind of discussion to the dev list
	add a comment to the issue
	consider that this was a first patch
	find a nicer way to improve
	no one should be implementing their own sorting code
	The interface is actually called Comparable (no "I")
	the approach sounds reasonable
	I find not safe writing our own sorting algorithm.
	`Modified: trunk/src/argouml-app/src/org/argouml/uml/generator/ui/ClassGenerationDialog.java`
	`Patch for ordering language names alphabetically in "generate all classes" window`
	`ClassGenerationDialog`
	`TODO: Get saved default directory`

6.

Email id	etnen46n3k7i2fnl
Email link	`http://markmail.org/message/etnen46n3k7i2fnl`
Email type	Entirely JUnit tests apart from very few lines of natural language
Email length	6 pages
Extraction type	sentences
Time to summarize	6 minutes

Extracted sentences	there is a report
	Was it me who broke this?
	Do I have to do something?
	`adding method field to operation properly panel`
	`extending model API with Artifact and Manifestation related methods`
	`“dummy" implementation for Artifact and Manifestation related methods`
	`implementation for Artifact and Manifestation related methods`
	`Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 21.025 sec`
	`Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 19.648 sec`
	[other 29 similar]
	Preparing the build directories in argouml-app
	Compiling the sources in argouml-app

B Benchmark creation test

B.1 "Round 1" material

```
PRE-TEST QUESTIONNAIRE

= Personal Data =
First name: [your name]
Last name: [your last name]
Age: [your age]


= Informatics experience =
Universities: [faculties you attended and where you are currently enrolled]

Known programming languages: [all programming languages you know, including
web and database languages]

Working experiences: [a list of any job related to informatics you did,
including realization of websites]
```

Figure 17. Pre-test form

```
POST TEST QUESTIONNAIRE

Name: [your first and last name]

1) How difficult was it to summarize emails by sentences?
###answer with a number from 1="very easy" to 5="very hard"###
System A, Thread 1 = []
System A, Thread 2 = []
System B, Thread 1 = []
System B, Thread 2 = []

2) How difficult was it to divide sentences into essential and optional?
###answer with a number from 1="very easy" to 5="very hard"###
System A, Thread 1 = []
System A, Thread 2 = []
System B, Thread 1 = []
System B, Thread 2 = []

3) How large was the amount of project-specific terminology used in the threads?
###answer with a number from 1="none used" to 5="very large"###
System A, Thread 1 = []
System A, Thread 2 = []
System B, Thread 1 = []
System B, Thread 2 = []

4) How difficult was it to understand the real meaning of the emails?
###answer with a number from 1="very easy" to 5="very hard"###
System A, Thread 1 = []
System A, Thread 2 = []
System B, Thread 1 = []
System B, Thread 2 = []

5) How much did you already know about the discussed system?
###answer with a number from 1="nothing" to 5="I am expert of it"###
System A = []
System B = []

6) How useful was the description about the system to understand the emails?
###answer with a number from 1="not, at all" to 5="very much"###
System A = []
System B = []

7) (optional) What were you thinking while completing this task? Have you got any
useful idea you would like to express regarding this experience? [fell free write
what you want]
```

Figure 18. Post-test questionnaire

Test guideline

Background and Goal

Programmers use *development emails* a lot. In these emails they discuss about a number of different things, such as how to fix bugs, ideas, or plans of future improvement. For this reason, these emails contain much useful information; they can be used even to understand a software system and its history!

Development emails, however, contain a "bit" too much information and developers read and write so many of them that the resulting big data is very hard to deal with. To reduce this problem, we are trying to create a method to *automatically* generate summaries of development emails. Cool, uh? But we must be sure that our method works! So we have to compare what our method produces to real full-fledged summaries, which only a smart human like you can create, by actually reading the email. That is why we need your help!

Your task is super simple: You read four email threads (which involves some programmers working on a software system) and, for each thread, you tell us what are the most relevant sentences (those that you would consider as a summary). That is it! Please just try to do your best in understanding the content of the threads and selecting the right sentences: This has a great impact on our work. If you like fancy terms, you are going to do *extractive summaries*, which do not require you to write anything (those would be *abstractive summaries*): You only select relevant information that is already in the conversation.

Summarization Procedure

Your task consists in creating an extractive summary of four email threads (*i.e.*, conversations). You *must* proceed in the following way (please, do not switch any step, we need you to do them in order):

I. Unzip the archived **task** file we sent you.

II. Read and fill in the **PRE_test_questionnaire** file.

III. Read the file **system_A/description** to have an idea of the system discussed in the first couple of email threads.

IV. Take (maximum) *45 minutes* to complete the following steps:

   (a) Read the discussion in **system_A/thread_1**.

   (b) From the original ones, **select ca. 30% sentences** (quoted sentences that are repeated in multiple emails do not count, just consider 30% of new sentences), which you think enclose important informations on the topic discussed in the thread.

   (c) Selecting 30% of the original sentences means that some of them will be more important, others less. For this reason, classify each chosen sentence as either **essential** or **optional**.

   (d) Read the discussion in **system_A/thread_2** and repeat steps (b) and (c).

V. Read the file **system_B/description** to have an idea of the system discussed in the second couple of email threads.

VI. Take other (maximum) 45 minutes to complete the same steps (a-d) on **system_B/thread_1** and **system_B/thread_2**.

VII. Read and fill the **POST_test_questionnaire** file.

Notes:

- Keep in mind that your aim should be to select sentences in a way that, by reading the generated summary, the readers do not need to refer to the original emails in most cases.
- Please, respect the time limit of **45 minutes** per system. The reason is that you can stay concentrated and produce better summaries.
- If you are in contact with other people involved in this test, please do not share opinions or summaries. This allows us to consider your summaries as a real *gold* reliable set.

Your output

What we need back from you is a zip file named as `[your_surname].zip` containing:

- The **PRE_test_questionnaire** filled.
- The **POST_test_questionnaire** filled.
- The **summaries of the email threads**. For what concerns these last ones, you can choose any format you want. Our suggestion is to highlight the sentences directly in the pdf files with a tool such as *Preview*: This is the fastest way. Please, just make sure to state clearly which color you used for essential sentences and which one for the optional ones.

Have fun with this task and thank you again!

Table 8. README file

B.2 “Round 2" post-test questionnaire charts (System A = FreeNet, System B = ArgoUML)

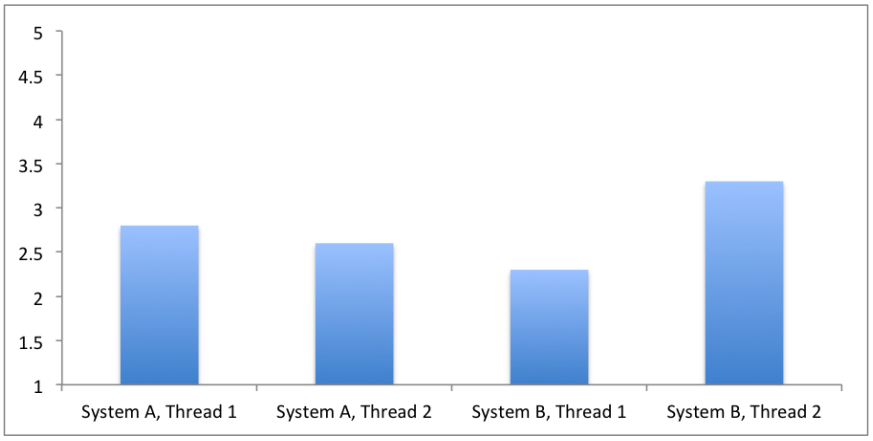

Figure 19. Average on answers to question “How difficult was it to summarize emails by sentences?" [1 = “very easy", 5 = “very hard"]

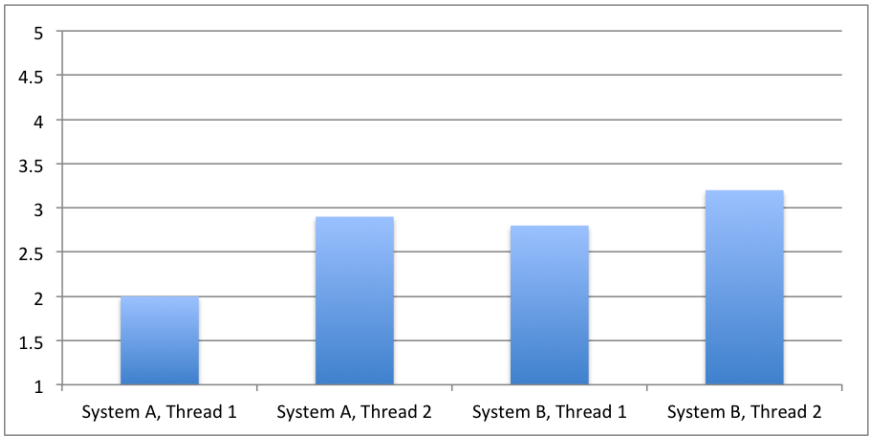

Figure 20. Average on answers to question “How difficult was it to divide sentences into essential and optional?" [1 = “very easy", 5 = “very hard"]

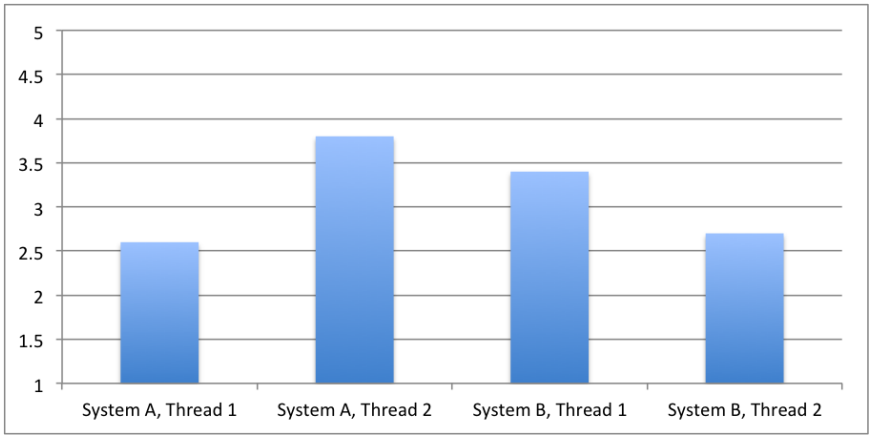

Figure 21. Average on answers to question “How large was the amount of project-specific terminology used in the threads?" [1 = “none used", 5 = “very large"]

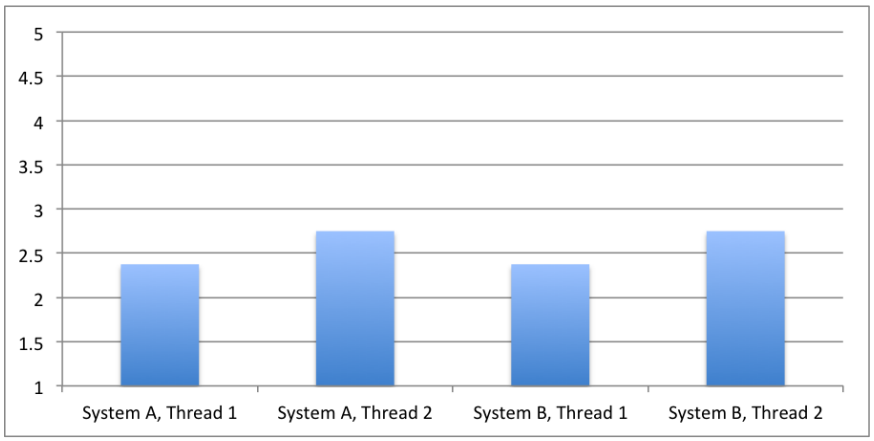

Figure 22. Average on answers to question “How difficult was it to understand the real meaning of the emails?" [1 = “very easy", 5 = “very hard"]

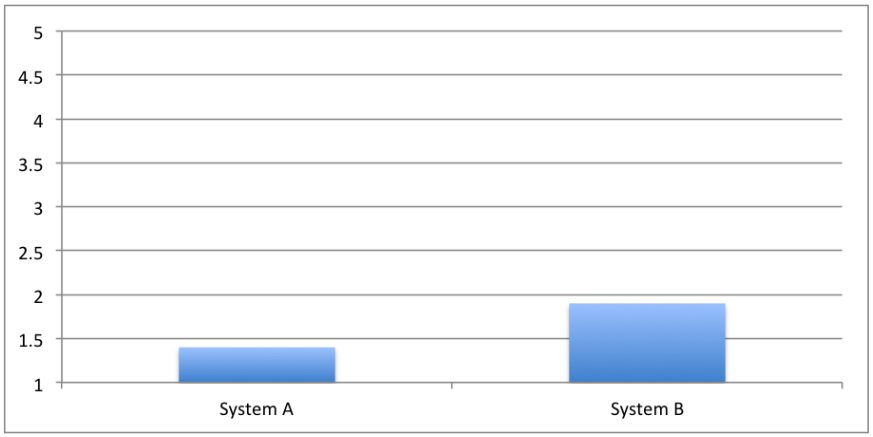

Figure 23. Average on answers to question “How much did you already know about the discussed system?" [1 = “nothing", 5 = “I am expert of it"]

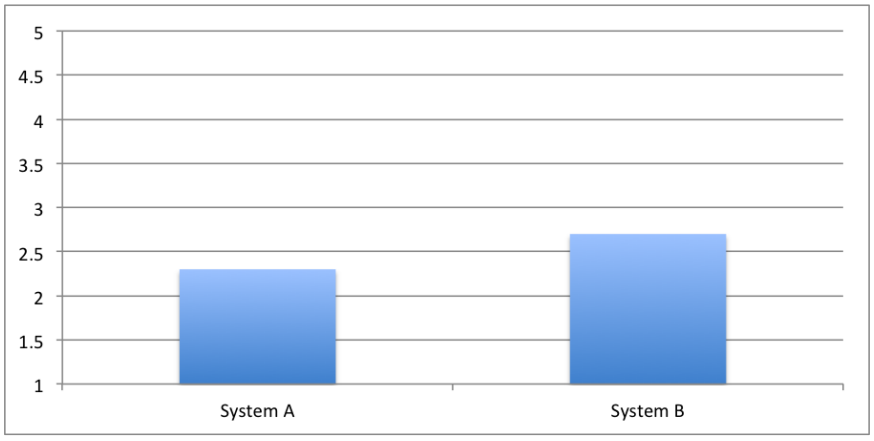

Figure 24. Average on answers to question "How useful was the description about the system to understand the mails?" [1 = "not, at all", 5 = "very much"]

B.3 Benchmark

A	B	C	D	E	F	G	H	I
author	sentence	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	abs_pos_nor
1	1	0	0	0	0	0	0	0.01351351
1	2	2	2	2	2	2	2	0.02702703
1	3	0	0	0	2	0	2	0.04054054
1	4	2	1	2	0	0	1	0.05405405
1	5	0	0	2	0	0	2	0.06756757
1	6	1	2	0	2	0	2	0.08108108
1	7	0	1	0	2	0	2	0.0945946
1	8	0	0	0	0	0	2	0.10810811
1	9	0	0	0	0	0	0	0.12162162
1	10	0	0	0	0	0	0	0.13513514
2	11	0	0	0	0	0	0	0.14864865
2	12	2	2	2	2	0	2	0.16216216
2	13	0	0	0	2	2	1	0.17567568
2	14	2	1	1	0	0	2	0.18918919
2	15	0	1	0	0	0	0	0.2027027
2	16	0	0	0	0	0	0	0.21621622
2	17	0	0	0	0	0	0	0.22972973
1	18	0	0	0	0	0	0	0.24324324
1	19	0	0	0	0	0	0	0.25675676

Figure 25. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns A-I)

I	J	K	L	M	N	O	P	Q
abs_pos_nor	chars	first_auth	first_auth_ar	first_no_gre	is_first	is_last	is_question	greet
0.01351351	12	1	2	0	1	0	0	1
0.02702703	87	1	2	1	0	0	0	0
0.04054054	104	1	2	0	0	0	0	0
0.05405405	61	1	2	0	0	0	0	0
0.06756757	27	1	2	0	0	0	0	0
0.08108108	88	1	2	0	0	0	0	0
0.0945946	106	1	2	0	0	0	0	0
0.10810811	43	1	2	0	0	0	1	0
0.12162162	7	1	2	0	0	0	0	0
0.13513514	5	1	2	0	0	1	0	0
0.14864865	31	0	1	1	1	0	0	0
0.16216216	107	0	1	0	0	0	0	0
0.17567568	67	0	1	0	0	0	0	0
0.18918919	114	0	1	0	0	0	0	0
0.2027027	28	0	1	0	0	0	0	0
0.21621622	7	0	1	0	0	0	0	1
0.22972973	4	0	1	0	0	1	0	0
0.24324324	6	1	2	0	1	0	0	1
0.25675676	66	1	2	0	0	0	0	0

Figure 26. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns I-Q)

Q	R	S	T	U	V	W	X	Y
greet	last_auth_ar	lines_answrc	link	num_adjects	num_answer	num_consnt	num_nouns_	num_nums
1	2	0	0	0	1	0.53846154	1	0
0	2	0	0	0	1	0.45454546	0.30769231	0
0	2	0	0	0.0952381	1	0.49193548	0.38095238	0.01612903
0	2	0	0	0	1	0.39726027	0.38461539	0.10958904
0	2	0	0	0	1	0.45454546	0.28571429	0
0	2	0	0	0.125	1	0.48514852	0.1875	0
0	2	0	0	0	1	0.51219512	0.16666667	0
0	2	0	0	0	1	0.46153846	0.3	0
0	2	0	0	0	1	0.71428571	1	0
0	2	0	0	0	1	0.6	1	0
0	1	10	0	0	1	0.54054054	0.42857143	0
0	1	10	0	0.05	1	0.52380952	0.4	0
0	1	10	0	0.13333333	1	0.48148148	0.2	0
0	1	10	0	0.11111111	1	0.47142857	0.18518519	0
0	1	10	0	0.33333333	1	0.5	0.16666667	0
1	1	10	0	0	1	0.71428571	1	0
0	1	10	0	0	1	0.5	1	0
1	2	7	0	0	1	0.42857143	1	0
0	2	7	0	0.06666667	1	0.48101266	0.2	0

Figure 27. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns Q-Y)

Y	Z	AA	AB	AC	AD	AE
num_nums	num_stopw_	num_verbs_	num_vocals_	num_words	quots	rel_pos_norr
0	0	0	0.30769231	2	0	0.1
0	0.38461539	0.15384615	0.35353535	13	0	0.2
0.01612903	0.28571429	0.14285714	0.32258065	21	0	0.3
0.10958904	0.15384615	0.15384615	0.26027397	13	0	0.4
0	0.42857143	0.14285714	0.33333333	7	0	0.5
0	0.4375	0.25	0.31683168	16	0	0.6
0	0.44444444	0.27777778	0.30081301	18	0	0.7
0	0.6	0.2	0.34615385	10	0	0.8
0	0	0	0.14285714	1	0	0.9
0	0	0	0.4	1	0	1
0	0.57142857	0	0.27027027	7	0	0.14285714
0	0.35	0.1	0.31746032	20	0	0.28571429
0	0.53333333	0.2	0.33333333	15	0	0.42857143
0	0.55555556	0.22222222	0.33571429	27	0	0.57142857
0	0.66666667	0.16666667	0.3125	6	0	0.71428571
0	0	0	0.28571429	1	0	0.85714286
0	0	0	0.25	1	0	1
0	0	0	0.28571429	2	0	0.02040816
0	0.46666667	0.2	0.3164557	15	0	0.04081633

Figure 28. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns Y-AE)

AE	AF	AG	AH	AI	AJ
rel_pos_norr	short_capt	subj_words_	hmn_score	binary_score	class
0.1	0	0	0	0	non-relevant
0.2	0	0	12	1	relevant
0.3	0	0	4	1	relevant
0.4	0	0	6	1	relevant
0.5	0	0	4	1	relevant
0.6	0	0.0625	7	1	relevant
0.7	0	0.11111111	5	1	relevant
0.8	0	0	2	0	non-relevant
0.9	0	0	0	0	non-relevant
1	0	0	0	0	non-relevant
0.14285714	0	0	0	0	non-relevant
0.28571429	0	0.05	10	1	relevant
0.42857143	0	0.06666667	5	1	relevant
0.57142857	0	0.03703704	6	1	relevant
0.71428571	0	0	1	0	non-relevant
0.85714286	0	0	0	0	non-relevant
1	0	0	0	0	non-relevant
0.02040816	0	0	0	0	non-relevant
0.04081633	0	0	0	0	non-relevant

Figure 29. Partial view of joined ArgoUML + FreeNet benchmark (rows 1-20, columns AE-AJ)

References

[1] A. Bacchelli, M. Lanza, and V. Humpa. Rtfm (read the factual mails) - augmenting program comprehension with remail. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 15–24, Washington, DC, USA, 2011. IEEE Computer Society.

[2] G. Carenini, R. T. Ng, and X. Zhou. Summarizing emails with conversational cohesion and subjectivity. In *Proceedings of ACL-08: HLT*, pages 353–361, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[4] R. T. N. Giuseppe Carenini and X. Zhou. Summarizing email conversations with clue words. In *In Proc. of ACM WWW 07*, 2007.

[5] S. Guo and S. Sanner. Probabilistic latent maximal marginal relevance. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 833–834, New York, NY, USA, 2010. ACM.

[6] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 223–226, New York, NY, USA, 2010. ACM.

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

[8] G. Murray. Summarizing spoken and written conversations. In *In Proc. of EMNLP 2008*, 2008.

[9] G. Murray, S. Renals, J. Carletta, and J. Moore. Evaluating automatic summaries of meeting recordings. In *in Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, Workshop on Machine Translation and Summarization Evaluation (MTSE), Ann Arbor*, pages 39–52. Rodopi, 2005.

[10] 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.

[11] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 505–514, New York, NY, USA, 2010. ACM.

[12] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.

[13] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[14] J. Ulrich and G. Murray. A publicly available annotated corpus for supervised email summarization, 2008.