Content Classification of Development Emails

Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, Michele Lanza
REVEAL @ Faculty of Informatics — University of Lugano, Switzerland

***Abstract*—Emails related to the development of a software system contain information about design choices and issues encountered during the development process. Exploiting the knowledge embedded in emails with automatic tools is challenging, due to the unstructured, noisy and mixed language nature of this communication medium. Natural language text is often not well-formed and is interleaved with languages with other syntaxes, such as code or stack traces.**

We present an approach to classify email content at line level. Our technique classifies email lines in five categories (*i.e.,* text, junk, code, patch, and stack trace) to allow one to subsequently apply ad hoc analysis techniques for each category. We evaluated our approach on a statistically significant set of emails gathered from mailing lists of four unrelated open source systems.

***Keywords*-Empirical software engineering; Unstructured Data Mining; Emails**

I. Introduction

Software repositories supply information useful for supporting software analysis and program comprehension [17]. Different repositories offer different perspectives on systems: For example, issue reports open a view on defective entities, thus enabling studies on defect location and prediction [14], while repositories archiving communication occurred among developers contain valuable "information [that helps] developers resolve crucial questions about the history, rationale, and future plans for source code" [36]. In particular, development emails contain discussions on topics ranging from implementation details to high-level design. The main maintainer of the Real-Time Linux explains that "mailing list archives provide a huge choice of technical discussions" and that developers do not write documentation because they believe that "it is all documented in the [...] mailing list" [18]. Email data helps to understand a system, its history, and its design rationale, thus supporting program comprehension [4]; email data also "offers the best opportunity for a researcher to observe the development process," as "developers reveal their thought processes most naturally when communicating with other software developers" [30].

However, obtaining objective and accurate results from software repositories is not trivial: The information extracted must be *relevant*, *unbiased*, and its contribution *comprehensible*. Researchers are studying repositories to understand what information is more relevant (*e.g.,* in bug reports [37] or in code changes [22]) and the impact of data quality on mining approaches and analyses [10], [34]. In particular, extracting valuable data from communication repositories (*e.g.,* IRC chat logs, mailing lists) require the most care, as the documents leave complete freedom to the authors. For example, Bettenburg *et al.* presented the risks of using email data without a proper cleaning pre-processing phase [9].

NL documents are usually treated as *bags of words*–a count of terms' occurrences. This simplification is proven to be effective in the information retrieval (IR) field, where techniques are tested on well-formed NL documents [25]. In software engineering, although effective for some tasks (*e.g.,* traceability between documents and code [1]), this approach reduces the quality, reliability, and comprehensibility of the available information, as NL text is often not well-formed and is interleaved with languages with different syntaxes: code fragments, stack traces, patches, *etc.*

We present a work for advancing the analysis of the contents of development emails. We argue that we should not create a single bag with terms indiscriminately coming from NL parts, code fragments, email signatures, patches, *etc.* and treat them equally. We need to recognize every language in an email to enable techniques exploiting the peculiarities of each category. This work contributes to a deeper and more detailed analysis of email communication among developers.

We propose an approach, based on a combination of parsing techniques and machine-learning (ML) methods, to classify the contents of development emails in five categories: NL, source code, patch, stack trace, and junk (text that does not add valuable information, such as auto-generated disclaimers, authors' signatures, or erroneous characters). Our technique works at the line level, which—by inspecting hundreds of emails—we found to be the appropriate granularity for email content classification. We created a web application to manually classify email content in the chosen categories. We classified a statistically significant set of emails from four JAVA open source software (OSS) systems, used to evaluate the accuracy of our approach.

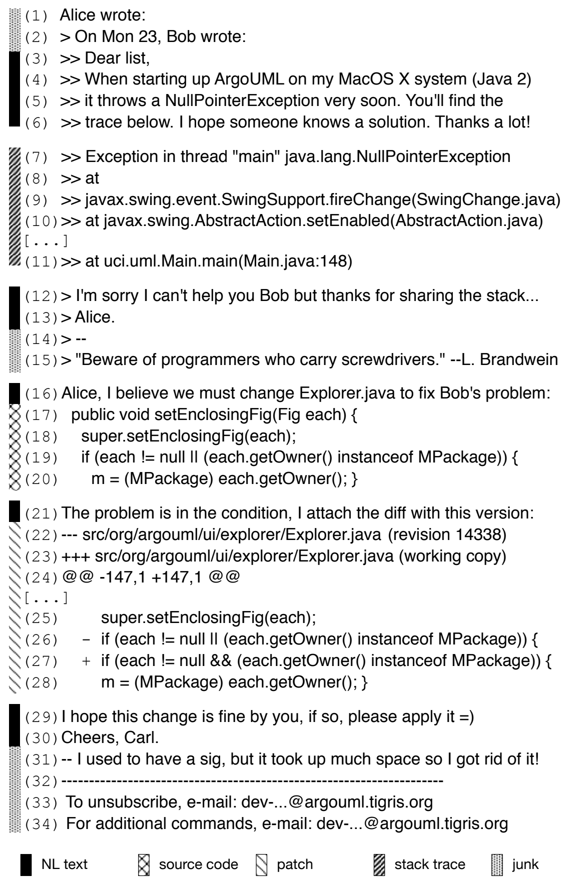
The contributions of this paper are:

1) a novel approach that fuses parsing and ML techniques for classification of email lines;
2) a web application to manually classify email content;
3) the manual classification of a statistically significant sample set of emails (for a total of 67,792 lines) from mailing lists of four different software systems–in the form of a freely available benchmark; and
4) the empirical evaluation of our approach against the benchmark.

Structure of the paper. In Section II we motivate our work. In Section III we describe the related work. In Section IV we show how we collected and manually annotated the email data. In Section V we detail our classification methods and their evaluation. We discuss threats to validity in Section VI and conclude in Section VII.

II. Motivation

Figure 1 shows the body of an example development email. Due to the variety of languages used, if we consider the content of such email as a single bag of words, we would obtain a motley set of flattened terms without a clear context, thus severely reducing quality and amount of available information. Inversely, by automatically distinguishing the parts composing the email, we support many tasks, such as:



```
(1)  Alice wrote:
(2)  > On Mon 23, Bob wrote:
(3)  >> Dear list,
(4)  >> When starting up ArgoUML on my MacOS X system (Java 2)
(5)  >> it throws a NullPointerException very soon. You'll find the
(6)  >> trace below. I hope someone knows a solution. Thanks a lot!

(7)  >> Exception in thread "main" java.lang.NullPointerException
(8)  >> at
(9)  >> javax.swing.event.SwingSupport.fireChange(SwingChange.java)
(10) >> at javax.swing.AbstractAction.setEnabled(AbstractAction.java)
[...]
(11) >> at uci.uml.Main.main(Main.java:148)

(12) > I'm sorry I can't help you Bob but thanks for sharing the stack...
(13) > Alice.
(14) > --
(15) > "Beware of programmers who carry screwdrivers." --L. Brandwein

(16) Alice, I believe we must change Explorer.java to fix Bob's problem:
(17)   public void setEnclosingFig(Fig each) {
(18)     super.setEnclosingFig(each);
(19)     if (each != null || (each.getOwner() instanceof MPackage)) {
(20)       m = (MPackage) each.getOwner(); }

(21) The problem is in the condition, I attach the diff with this version:
(22) --- src/org/argouml/ui/explorer/Explorer.java  (revision 14338)
(23) +++ src/org/argouml/ui/explorer/Explorer.java (working copy)
(24) @@ -147,1 +147,1 @@
[...]
(25)        super.setEnclosingFig(each);
(26)     -  if (each != null || (each.getOwner() instanceof MPackage)) {
(27)     +  if (each != null && (each.getOwner() instanceof MPackage)) {
(28)          m = (MPackage) each.getOwner(); }

(29) I hope this change is fine by you, if so, please apply it =)
(30) Cheers, Carl.
(31) -- I used to have a sig, but it took up much space so I got rid of it!
(32) ------------------------------------------------------------------------
(33)  To unsubscribe, e-mail: dev-...@argouml.tigris.org
(34)  For additional commands, e-mail: dev-...@argouml.tigris.org
```

█ NL text ▨ source code ▧ patch ▨ stack trace ▒ junk

Figure 1. Example development email with mixed content

Traceability recovery. In Figure 1, the email is referring to several classes (*e.g., Main*, *Fig*, and *MPackage*), but only the class *Explorer* is critical to the discussion: It causes the failure and the email's author is changing it to provide a solution. We realize the importance of *Explorer* by reading the NL line 16. As part of our ongoing investigation on email archives [4], we often found this pattern: Artifacts mentioned in NL parts of emails are more relevant to the discussion than artifacts mentioned in other contexts (*e.g.,* stack traces). A traceability method based on bags of words (*e.g.,* [5]) cannot recognize whether references to artifacts appear in a NL context, to increase the link relevance. Such a method can only use the number of occurrences to weight more certain terms [25], leading to imprecise results. In Figure 1, a weighting based on occurrences would give the most relevance to class *MPackage* (mentioned 5 times), which is actually marginal to the discussion.

By recognizing the context in which a term appears, one can elicit weights for words appearing in a document dynamically and more accurately, improving the traceability links's quality and giving more information to the user.

Stop words removal. To better characterize documents, IR research invites to remove *stop words*, *i.e.,* very common words [25], thus weighting more the peculiar terms of a document. This approach is less beneficial when applied to development emails: By removing stop words, one reduces the noise in NL parts, but also deletes information in parts with a different vocabulary (*e.g.,* source code). For example, deleting the stop word "each" from the content of Figure 1 means also deleting a variable name in a code fragment (lines 17–20) and a patch (25–28). This is suboptimal, since variable names provide relevant information [23]. Similarly, we delete important information by removing programming language keywords from NL.

By recognizing the different parts that compose an email, one can use different common terms removal techniques, exposing the most relevant information.

Artifact summarization. Due to the amount of data produced during a system's evolution, researchers investigated how to expose only the significant parts to reduce information overload (*e.g.,* [28]). The proposed techniques are tailored to specific types of artifact (*e.g.,* code [19], NL documents [20]) and cannot be applied to mixed documents, such as emails.

By recognizing the different parts of an email, one can use the most suited summarization technique according to each part's type and extract correct information.

Fact extraction. To know the facts expressed in code fragments, patches, or stack traces, one can use ad hoc parsers. In Figure 1, using a parser for patches, one recognizes that the file being modified is *Explorer* (lines 22–23). Similarly, NL text can be analyzed with NLP techniques [21]. However, ad hoc parsers cannot be applied to mixed content, as they are not robust enough to manage unexpected data.

By distinguishing the type of each email line, we can exploit ad hoc analysis techniques to extract precise information.

Non-essential information removal. In Figure 1, 8 lines out of 34 contain irrelevant data–"junk". Previous research indicated how some changes in version history are not essential, and how their detection and filtering can improve change-based analysis techniques [22]. Similarly, the detection and removal of junk from email content increase the quality of the data [9], thus improving the quality of analyses.

By recognizing the noise in emails, the important data emerges, improving the information extraction quality.

III. Related Work

Researchers applied NL analysis techniques to software-related documents and devised approaches to improve the comprehension of the NL parts. For example, Dekhtyar *et al.* [15] discussed the promises and perils of text mining for NL software artifacts. Here we focus on research on the recognition of the different parts that compose NL artifacts.

The work by Bettenburg *et al.* [9] focuses on making the research community aware of the noise in email data and presents the importance of a proper cleaning pre-processing phase. The authors suggest possible filtering heuristics to recognize noise and irrelevant information. Later, Bettenburg *et al.* devised infoZilla, a tool to recognize and extract patches, stack traces, source code snippets, and enumerations in the textual descriptions that accompany issue reports [8]. It is composed of four independent filters, one per category, which are used in cascade to process the text. The source code filter exploits an approach inspired by island parsing [27], while the others are based on text matching implemented through regular expressions. In the task of differentiating documents (*i.e.,* deciding whether they contain or not each category), infoZilla reached almost perfect results, with precision and recall values above 0.95 in all the categories. InfoZilla has been effectively applied to investigate relevant features of text in issue reports [37].

Compared to bug comments, development emails present the following differences: (1) they contain a larger NL vocabulary, since the discussion is not limited to bug related issues; (2) they present more noise, generated for example by email headers and authors' signatures; and (3) emails pose greater challenges in text recognition, since many email clients automatically wrap long lines of text, thus breaking the right formatting [11]. Bird *et al.* proposed an approach to measure the acceptance rate of patches submitted via email in OSS projects [11]. They extracted code patches from emails and used them to analyze the developers' interactions.

Some information retrieval approaches targeted the classification of text or the recognition of information with specific patterns [21], exploiting probabilistic and ML models (*e.g.,* Maximum Entropy Models [7] or Hidden Markov Models [6]). Tang *et al.* addressed the issue of cleaning the email data for text mining [33]. The authors proposed a four-step approach to clean emails: (1) non-NL text filtering, (2) paragraph recognition, (3) sentence boundaries detection, and (4) word normalization. Their method first filters out email headers, signatures, and program code (*without* a distinction from patches or stack traces); then it recognizes the paragraphs and sentences that compose the remaining NL text; finally, it corrects misspelled words. The authors randomly chose a total of 5,459 emails from 14 unrelated sources (*e.g.,* newsgroups at Google) and created 14 data sets in which they manually labeled headers, signatures, quotations, and program codes. Given the labelled data, the authors implemented a classifier for each step of their approach. All the classifiers use Support Vector Machines (SVM) and are based on specific features (*e.g.,* number of words). At line level classification, they achieved an f-measure of 0.81 in recognizing code, and 0.98 and 0.90 for header and signature.

Carvalo and Cohen devised methods to recognize signature blocks and reply lines in emails [13]. They worked at the line level and tested the effectiveness of a set of features with many ML classifiers. In the signature detection task the methods reached an f-measure value of 0.97.

In our previous work we proposed Besc, a *lexical* approach to recognize the lines of development emails that contain Java code fragments [3]. Even though Besc achieves good results in terms of effectiveness and practical performance, it specifically focuses on recognizing code and can only be partially used in the context of a more comprehensive email text classification. For example, Besc merges lines of stack traces, patches, and actual source code, under the umbrella of code fragments: In Figure 1, it would indiscriminately recognize lines 9–11, 17–20, and 25–28 as code fragments. Although such an approach can be useful for certain system analyses (see [3]), it generates a classification that does *not* allow one to (1) distinguish lines written in NL; (2) recognize patch context and headers (*e.g.,* lines 22–24 in Figure 1); (3) distinguish complete blocks of stack traces, (*e.g.,* lines 7–11, to use ad hoc parsers); (4) remove the non-relevant information (*i.e.,* "junk").

Summing up, previous work differs from the current as it:

- addressed more compact classification tasks, for example only detecting patches [11] or signatures [13];
- considered a larger granularity or different data sources (*e.g.,* bug reports [8]);
- did not distinguish structured data forms (*e.g.,* by merging patches, code, and stack traces [3], [33]);
- hard-code all the classification rules, thus not covering unexpected cases (*e.g.,* [3]).

We strive for an approach with a fine granularity and a wide breadth, able to provide a robust classification, which can be used for increasing the quality of subsequent analyses.

IV. Data Collection and Classification

Since we strive for devising a method for reliably and precisely classifying email lines, with the aim of improving data quality and comprehension, we need data sets that are *accurate*, *comprehensive*, and of *statistically significant* sizes. This is critical for the validation and leads to more reliable training for the supervised classification methods. To this aim, we implemented a web application to assist the manual classification of email content in categories.

A. Data Collection

Different software systems often use different applications to manage email repositories. We tackled this issue by importing data from MarkMail (http://markmail.org), a web service storing more than 8,000 up-to-date mailing lists.

Table I
Email data sets used in the experiment, by system

System URL	Mailing list			
	Inception	Emails		
		Total	After Filtering	Sample
ArgoUML *argouml.tigris.org*	Jan 2000	25,538	25,538	379
Freenet *freenetproject.org*	Apr 2000	23,134	23,134	378
JMeter *jmeter.org*	Jan 2006	24,005	5,814	361
Mina *org.apache.mina.dev*	Feb 2001	21,384	14,499	375
Total		**94,061**	**68,985**	**1,493**

Table I shows the four software systems and mailing lists we considered. We selected unrelated systems emerging from the context of different free software communities, *i.e.,* Apache, ArgoUML, and Freenet. The development environment and paradigms, and the usage of the mailing lists are likely to differ, thus mitigating external validity threats.

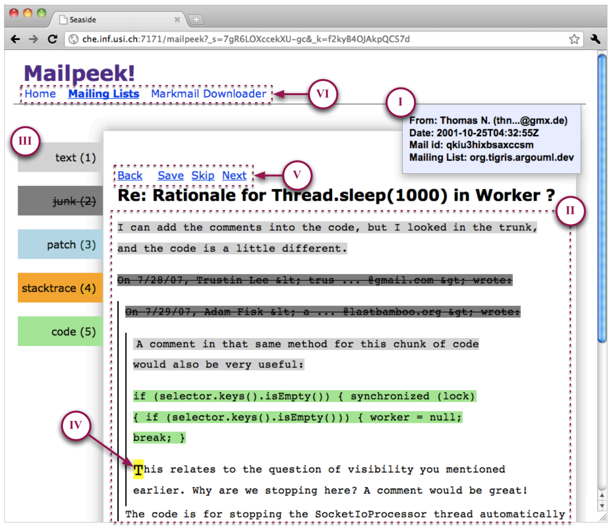
We imported all the messages starting from the mailing list inception (second column in Table I) to the end of 2010. The only pre-processing conducted on the emails was filtering out messages automatically generated by the bug tracking system and the versioning system.

From each filtered mailing list, we extracted statistically significant sample sets (last column, Table I), which were used by the approach without any pre-processing on the text. Since we had no prior knowledge on the distribution of line categories in the populations, we opted for simple random sampling [35] to pick the emails. The chosen sizes have a 95% confidence level[1] and a 5% error margin.

B. Data Classification

To test our approach and train supervised ML classifiers, we needed to manually classify the 1,493 sample emails. To ease this manual task and alleviate its error-proneness, we devised Mailpeek, a web application written in Smalltalk using the Seaside framework [16]).

[1]See [5], [35] for more information about sample size determination.



From: Thomas N. (thn...@gmx.de)
Date: 2001-10-25T04:32:55Z
Mail id: qkiu3hixbsaxccsm
Mailing List: org.tigris.argouml.dev

Re: Rationale for Thread.sleep(1000) in Worker ?

```
I can add the comments into the code, but I looked in the trunk,
and the code is a little different.
```

~~On 7/28/07, Trustin Lee &lt; trus ... @gmail.com &gt; wrote:~~

~~On 7/29/07, Adam Fisk &lt; a ... @lastbamboo.org &gt; wrote:~~

```
 A comment in that same method for this chunk of code
would also be very useful:

if (selector.keys().isEmpty()) { synchronized (lock)
{ if (selector.keys().isEmpty())) { worker = null;
break; }

This relates to the question of visibility you mentioned
earlier. Why are we stopping here? A comment would be great!
The code is for stopping the SocketIoProcessor thread automatically
```

Figure 2. Mailpeek: our web app for classifying email content

Figure 2 shows the main window of Mailpeek, as it appears in a web browser after a user selects a mailing list of interest and the application extracts a random email among those not automatically filtered. Mailpeek displays the email metadata (point I) and content (point II), with vertical bars to show indentation levels and increase readability.

Users conduct the classification task at the *character* level: To label a block, they (1) click on starting and ending characters, (2) verify the correctness of the selection (which is shown in a yellow background), and (3) apply the appropriate category, either by clicking on a button in the left menu (point III), or using keyboard shortcuts. The character granularity provided us the basis to decide which granularity was appropriate for the automatic classification, *i.e.,* line granularity (see Section IV-C).

When users hover with the mouse on any character in the email content area (point II), the character font size triples (point IV). According to Fitts' Law [24], this eases the selection, thus decreasing fatigue and errors.

Once an email is completely classified, the user clicks on *save* (point V) and Mailpeek loads another random email among those not yet classified. The *skip* link allows the user to leave out non-valid emails that were not removed by the filtering phase. The top menu (point VI) allows users to change mailing list or trigger the importer.

Two graduate students from the REVEAL Research Group at the University of Lugano, with several years of Java programming experience, conducted the manual classification task on two distinct sets of emails. We evaluated the inter-rater agreement by asking them to also classify 5% of the emails analyzed by the other person. In this sample, we found 12 non concordant lines (less than 0.2%).

C. Data Distribution

Table II reports categories' distributions in the sample sets.

Table II
Distribution of the categories per line, by system

		ArgoUML		Freenet		JMeter		Mina		Total	
NL Text	■	10,945	47.2%	7,923	59.6%	7,778	41.8%	6,496	51.2%	33,142	48.9%
Junk	▒	11,122	47.9%	4,096	30.8%	9,734	52.3%	4,633	36.5%	29,585	43.6%
Patch	▧	470	2.0%	986	7.4%	339	1.8%	287	2.3%	2,082	3.1%
Source Code	▩	304	1.3%	29	0.2%	591	3.2%	990	7.8%	1,914	2.8%
Stack Trace	▨	364	1.6%	254	1.9%	165	0.9%	286	2.3%	1,069	1.6%
Total		23,205		13,288		18,607		12,692		67,792	

Most lines are NL; more than 30% of lines are junk, thus stressing the impact of noise on email data; the frequency of other categories is lower and the ranking changes according to the mailing list. The different composition of the email sets' contents reflects the different usage of mailing lists among the communities. Some lines are *hybrid*: they belong to more than one category, and are mostly composed of junk not separated by the NL text. They account for less than 5% of the population (*i.e.,* 3,362 lines). To mitigate the bias in the experiment we include them as separated instances.

V. Experiment

We created a number of techniques based on ideas gathered both from the IR field, which we reshaped and adapted, and from language programming parsing. Even though the techniques can be used in isolation, we achieved the best results by creating a unified approach.

A. Term Based Classification

In IR systems, documents are considered as bags of words, where syntactic information, ordering, and constituency of the words play no role in determining their meaning. In practice each document is modeled as a vector of features, which correspond to *terms* in the corpus vocabulary. For example, if we consider a document (d), the cardinality of the vocabulary ($|C|$), and how many times each term (t_i) occurs in d, we could define the document vector as: $v_d = [t_{1(d)}, t_{2(d)}, \ldots, t_{C(d)}]$.

This simple vector modeling has been widely used with supervised ML algorithms to achieve very effective results in automatic text classification [25], [31]. We ground the first techniques on the same basis: We consider lines as vectors of terms and use ML for their classification.

In the following we we describe and motivate our choices in terms of the used ML technique and vector features (*i.e.,* terms), which cannot be based on results from the IR field, as they refer to other domains and classification tasks.

1) Machine-learning method: We employ Naïve Bayes, a method of *supervised* learning (*i.e.,* ML algorithms that use classified training examples to infer the classification function). Naïve Bayes relies on the *conditional independence* assumption: The presence of a feature is unrelated to the occurrence of the other features. Even though the assumption is a strong simplification, the method often outperforms more sophisticated techniques [21]; in particular in text classification, *Naïve Bayes* achieves significant results [12]. An asset of Naïve Bayes is its linear complexity, which allows training and classification to be performed efficiently, even with a very large number of features.

The method uses Bayes' rule [21] to compute the probability that a line l, made of t_k terms, belongs to class c:

$$P(c|l) \approx P(c) \prod_{k} P(t_k|c) \tag{1}$$

It computes the posterior probability $P(c_i|l)$ for each class and chooses the one with the highest probability. This is the *maximum a posteriori* (MAP) hypothesis [21]:

$$C_{MAP} = \arg\max_{c_j \in C} P(c|l) \approx \arg\max_{c_j \in C} P(c) \prod_{k} P(t_k|c) \tag{2}$$

If we want to classify the line d = "*Alice wrote* :" as *text*, *junk*, or *code*, the algorithm first computes the probabilities as: $P(text|l) = 0.43$, $P(junk|l) = 0.55$ and $P(code|l) = 0.02$, then selects the value 0.55, thus classifying l as *junk*.

2) Selection of the terms: **Words:** They are the fundamental tokens of all the languages we want to classify. We judge the words in our corpus of 67,792 non-empty lines to be proper features for line modeling. Contrarily to most IR methods, we do perform neither *stop word removal* (*i.e.,* excluding very common words), as we expect very frequent words to be representative of a class (*e.g.,* Java keywords in code), nor *stemming* (*i.e.,* collapsing the morphological variants of a word), as we expect some variants to be more characteristic of certain classes (*e.g.,* verb tenses in NL text).
Punctuation: We must distinguish lines written in languages with different syntaxes, thus we consider punctuation to be a valuable aspect. Unless the punctuation marks are separated by words or spaces (*e.g.,* the dots in *javax.swing.*, are two occurrences of the feature "."), we consider them as a single term, thus recognizing special characters, such as "@@" in line 24 in Figure 1. We do not consider email reply threading characters (*e.g.,* > and >> in lines 2-15 in Figure 1) at this point, as they do not have a definite role for line classification.
Bi-grams: Naïve Bayes relies on the conditional independence assumption, which makes the modeling of NL text features feasible. However, the other considered languages have a stricter syntax with patterns of terms appearing together (*e.g.,* "public void" in code). To model this dependency characteristic of some terms, thus also reducing the negative effects of Naïve Bayes' assumption, we also consider *bi-grams* (*i.e.,* pairs of terms appearing one after the other).
Context: All the features considered so far are extracted *only* from the line under classification. However, some of the considered classes (*i.e.,* patch and stack trace) have a structure recognizable only if considering surrounding lines. For example, line 18 and line 25 have the same content,

thus can be mapped to the correct class only considering the context lines. Researchers proposed to solve a similar problem by adding features with characteristics of lines close to the one under classification [13], [33]. We adapt this approach to our case by considering what appears in the preceding and following lines. For example, in addition to "@@", we have the features "@@-lineBefore", and "@@-lineAfter".

Table III
RESULTS WITH TERM BASED CLASSIFICATION, BY FEATURE SETS

	Number of Features	10-fold cross validation		Mailing list cross validation	
		Correct Lines	Impr. sig.	Correct Lines	Impr. sig.
Words	12,658	46,555 68.6%		46,056 67.9%	
Words, Punctuation	19,384	62,938 92.8%	p < 0.001	58,172 85.8%	p < 0.001
Words, Punctuation, Bi-grams	145,187	63,413 93.5%	p < 0.001	58,568 86.4%	p < 0.001
Words, Punctuation, Bi-grams, Context	435,561	63,708 93.9%	p < 0.001	60,580 89.4%	p < 0.001

3) Line modeling: After defining the aforementioned features, we modeled each line as vector a of $n+1$ dimensions. The first n elements are the chosen features, while the last one is the manual classification value (*e.g.,* "patch"). The first column of Table III shows the values of n according to the considered subset of features. Each feature is populated with the corresponding term's occurrences in the line.

B. Training and Testing

Since we use a supervised ML algorithm, we need to train it on classified data. We use two different approaches for training the model and show how this affects the results when testing of the model's accuracy. To evaluate the model's accuracy, we count the number of correctly classified lines and we use two IR metrics [25]: *precision* ($P = \frac{|TP|}{|TP+FP|}$) and *recall* ($R = \frac{|TP|}{|TOT|}$). TP (true positives) are correctly classified lines, FP (false positives) are not correctly classified lines, and TOT is the total number of lines. *F-measure* is the weighted harmonic mean of P and R [25].

1) Ten-fold stratified cross-validation: As a first step, we apply 10-fold stratified cross validation [35]: We split the dataset in 10 folds, use 9 folds (90% of the lines) to train the prediction model, and use the remaining fold to test the model's accuracy. This process is repeated 10 times rotating the training and testing folds. The distribution of classes is kept equal in training and test sets. Columns 2 and 3 in Table III show the results. Each subset of features adds information that increases the results in a significant way (column 3). When considering all the features, the ratio of correctly classified instances reaches almost 94%.

2) Mailing list cross-validation: Different mailing lists discuss about different systems and are likely to use different words and jargon. For example the mailing list signature (*e.g.,* lines 32–34 in Figure 1) have different terms in each mailing list. Thus, term-features that work for one mailing list may not be useful for others. To better test the generalizability of the results achieved by the classifier, we conduct a "mailing list cross validation." In practice, it is a 4-folds cross validation, in which folds are neither stratified nor randomly taken, but correspond exactly to the different mailing list: We train the classifiers on three mailing lists and we try to predict the classification of the remaining mailing list. We do this four times rotating the mailing lists and we measure the average results. Columns 4 and 5 in Table III show the results.

As expected, testing with mailing list cross validation, the performance of the classifier drops, even when considering all the features. However, this is a more relevant test to understand the results of the classifier applied to unseen JAVA development mailing lists, and we use it in following.

Table IV
MAILING LIST CROSS VALIDATION ON THE BEST SET OF FEATURES

classified as →		NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	■	32,062	1,046	20	8	6	0.894	0.967	0.929
Junk	▦	3,269	26,225	54	14	23	0.942	0.886	0.913
Patch	▧	207	343	946	585	1	0.452	0.454	0.453
Source Code	▩	309	121	1,074	410	0	0.403	0.214	0.280
Stack Trace	▨	35	97	0	0	937	0.969	0.877	0.920

Table IV reports confusion matrix [25], precision, recall, and F-measure values for the classification with all the term-features (*i.e.,* words, punctuation, bi-grams, and context). The best results are achieved in classifying text, junk, and stack trace, while patch and code are often misclassified among themselves. This is reasonable, since recognizing those lines requires a large context: Even a human reader could not determine to which class line 28 in Figure 1 belongs without inspecting many lines. However, differentiating code and patches might be useful for various tasks, such as improving traceability links or automatically estimating the topic and purpose of the email (see Section II).

C. Term Based Features and Overfitting

By considering the entire set of features (*i.e.,* words, punctuation, bi-grams, and context), we obtain a complex classification model with more features than training instances. In such a scenario, *overfitting* is likely to occur—this hypothesis is supported by the reduced performances of the classifier in mailing list cross validation (see Table III). By reducing the features that are not valuable to correctly predict instances outside the training set, we decrease overfitting and increase the generalizability of the results.

Since we use words and punctuation to describe the common traits of each language, we suppose that the terms that rarely occur in the corpus are less relevant and can be removed. We investigate this hypothesis by gradually filtering out features (from all four kinds) that appear in less than t lines and inspecting the results.

Figure 3 shows the average classifier's performance in mailing list cross validation, with t ranging from 1 to 4,587 (higher values reduce the number of features to less than 10

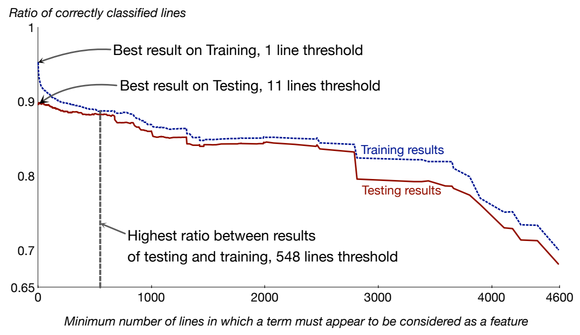


Figure 3. Results on training and test sets, by line threshold for features

greatly reducing the results). The blue dashed line (above) is the average percentage of correctly classified lines on the training set, while the red solid line (below) is the average percentage on the test set. The best result on the training set (*i.e.,* 96.1%) is set at *t* value of 1, (*i.e.,* we consider all the features, 115,864 on average when training on three mailing lists), while the best result on the testing set (*i.e.,* 89.9%) is set at *t* value of 11 (*i.e.,* 5,618 features on average), which reduces some noise. The optimal *t* value for the best testing set results, however, changes according to the mailing list: Two lists have a *t* value of 2, one of 25, and one of 46. A valid approach to find a good value for *t*, also for unseen data, is to consider the point with the highest ratio between testing results and training results [35]. We find this hot spot with a threshold of 548 lines (*i.e.,* 122 features on average). Interestingly the number of features is a tiny fraction of the initial ones, but the testing results are reduced only by a 1.5% (*i.e.,* 88.3%). Higher thresholds lead to lower performances.

D. Parsing Based Classification

We tackle the classification from a different perspective and use a different approach: parsing. In fact, three of the considered classes (*i.e.,* stack trace, patch, and source code), which are either produced or consumed by a machine, present a clearly structured and defined syntax that may be recognized even if embedded in a noisy unstructured context. We use a technique from our previous work [2] to write specialized parsers per each class (excluding NL text), based on the concept of island parsing [27]. For space reasons, we detail only the most salient features of each parser. The complete source code is available at http://mucca.inf.usi.ch.

D.1. Stack Trace Parsing

We define a terminology: The *exceptionMessage* is the NL message included at the beginning of stack traces (*e.g.,* line 7 in Figure 1); the *atLine* is a method invocation within a specific file (*e.g.,* lines 8–11); the *ellipsisLine* reduces lengthy stack traces and has the form: "…<number> more"; the *causedByLine* may be in any point in a trace and introduces a nested trace and has the form: "Caused by: <stacktrace>".

Among these elements, *atLine*s and *ellipsisLine*s are the most recognizable ones. By using the concept of *island parsing* [32] and the SMALLTALK parser generator *PetitParser* [29], we defined a grammar to obtain a parser to extract these two elements, even if embedded in the noisy content of emails or arbitrarily split on more lines, because of erroneous line breaks. By testing our approach on the whole corpus we found no errors in this parsing phase.

The *exceptionMessage* and the *causedByLine* elements have an unpredictable structure (*e.g.,* different JAVA virtual machine versions may output the same error message differently), thus they cannot be parsed with a specific grammar. We use a double-pass approach: First, we mark all the *atLine*s and *ellipsisLine*s, then we look for each line that contains strings such as "exception", "error", *etc.* When such a line exists, if the next *n* lines belong to those lines marked in the first step, we classify it and all the lines up to the first *atLine* as *stack trace*. Since *exceptionMessage*s are made of not more than three lines, we use an *n* of 3. The value can be adapted if a system uses another message length.

For example, when we apply our stack trace parser to the email in Figure 1, in the first pass, it classifies lines 8–11 as *stack trace*; in the second pass, it considers lines 5 and 7 as *exceptionMessage* candidates, since they both contain the string "exception". Finally, it only picks line 7, because in the next 3 lines there is an *atLine* (in this heuristic, we also count the empty lines, such as the line between 6 and 7).

D.2. Patch Parsing

We define a terminology: The first two lines of a patch (*e.g.,* lines 22 and 23) are the *patchHeader*, and contain the reference to the modified file and, optionally, the revision versions (*e.g.,* lines 22 and 23 in Figure 1); the lines showing the changes done by the patch (*e.g.,* line 22) are the *patchBlockHeader*; and all the lines in the chunk (*e.g.,* lines 25–28) are the *patchBlock*.

Similarly to the previous approach, we start from the most recognizable lines and expand to include the more ambiguous ones. The parsing is done in a single pass: We wrote a grammar to generate a parser that recognizes the *patchHeader* by using "—", "+++", and "@@" as hooks; then it recognizes the *patchBlockHeader* (thanks to its clear structure), then it matches the following *patchBlock*. The *patchBlock*s are problematic, since they have variable length and an unclearly defined ending. In fact, after the deleted and added lines (which are marked with initial "+" or "-" signs, as in lines 26 and 27), patches include *some* contextual lines: Their number may vary between zero and three, or more if not well formatted. We implemented a lookahead heuristic that checks whether the lines after the "+" or "-" signs might be good candidates as patch. The heuristic checks whether the lines are source code, through a reduced version of the code parser described later, and in the positive case it classifies them as *patch*.

D.3. Source Code Parsing

Among the three classes with the most structured language (*i.e.,* stack trace, patch, and source code), code is the most ambiguous. This is due to the fact that email authors usually do not report complete compilation units (*e.g.,* a whole JAVA class definition), but only selected fragments (*e.g.,* the method declaration in lines 17–20 in Figure 1). These fragments may present more ambiguities, with respect to NL and junk, than blocks of patches or stack traces: For example, if a line comprises only the words "public class", it can be either the beginning of a class declaration or a simple NL sentence.

We devised a parser based on the technique detailed in our previous work [2]: We wrote a complete JAVA grammar for PetitParser, by implementing the latest specification of the official JAVA language. Then, we implemented an island parser able to recognize most of the constructs of the grammar (including single and multi-line comments, but excluding constructs that are too ambiguous with NL, such as expressions), starting from the most comprehensive (*i.e.,* compilation unit) down to very specific ones (*e.g.,* expression statements). We also added rules to recognize incomplete constructs (such as method declarations without the body– common in email discussions).

Compared to BESC [3], this island parser reaches higher precision and is also able to locate constructs that span on more lines. For example, BESC cannot classify a line with only "public class" as code, while our island parser classifies it as code depending on the surrounding lines.

This parser matches most of the content of *patchBlock*s, as they also contain valid source code. This raises a number of false positives. We avoid this by chaining the code parsing to the patch parsing: First we detect the patches, then, on the lines that are *not* classified as patch, we use the code parser. As a beneficial side effect, this chained procedure reduces the text and the ambiguities to be managed by the island parser, thus increasing the performances.

D.4. Junk Parsing

Noisy text, such as authors' signatures, is hard to automatically distinguish from NL text; however, some peculiar common patterns can be matched with a parser. This approach is made of three steps: (1) matching and classification of email headers (*e.g.,* lines 1 and 2 in Figure 1) with a regular expression; (2) identification and extraction of signatures of mailing lists (*e.g.,* common lines added to the end of every email sent to the same list, such as lines 32–34) and authors; and (3) usage of the recognized signatures to automatically compose a grammar for generating a parser to match them, under any possible formatting or position in the email body. To recognize signatures, we consider all the emails whose last block of text is not quoted from previous emails (this can be easily achieved by considering lines that do not end with email reply threading characters, such as > and >> in lines 2-15 in Figure 1). In these emails, the authors themselves conclude the message and most probably include their signatures. For example, the email in Figure 1 contains the author's signature in the last block. Among the selected emails, we only consider the last not quoted block. We analyze it backward starting from the last line (*e.g.,* from line 34 up to 16). When we encounter a line that starts with, or is only composed of, two or more dashes, underscores, or stars, we take out the lines up to the bottom and consider this as a signature. The process continues until it reaches the top of the not quoted block. For example, the algorithm, applied to the email in Figure 1, would extract lines 32 to 34, and line 31 as block signatures.

By classifying these blocks as junk, we would miss the cases in which signatures are in quoted text (*e.g.,* lines 14–15). We, thus, conduct the third step: We use each extracted string to automatically define a grammar able to recognize the signature in any possible position or formatting the text; then, we use these grammars to automatically generate the relative parsers; finally we classify matched lines as *junk*.

D.5. Results

Table V reports the results of each parser in the classification of the lines into the corresponding type. For example, the first line covers the results in using the Stack trace parser to classify lines as *stack trace*. The false positives (*e.g.,* 4 in the first row) are lines classified as *stack trace* by the method, but with a different manual classification.

Table V
SINGLE CLASSIFICATION RESULTS ACHIEVED BY USING PARSERS

	Total Instances	True Positives	False Positives	Precision	Recall	F-Measure
Stack trace parser	1,069	1,054	4	0.996	0.986	0.991
Patch parser	2,082	1,996	0	1.000	0.959	0.979
Source code parser	1,914	1,715	74	0.959	0.896	0.926
Junk parser	29,585	20,372	226	0.989	0.689	0.812

All the parsers reach high classification values, while being mailing list independent and requiring no training. However, parsers have limitations: (1) They are manually implemented, and for this reason they cannot predict or cover all the possible variants of the patterns that they match, especially due to truncated content; (2) the values reached in classifying junk are lower than those achieved with the ML approach.

Next, we present a method that overcomes these issues by fusing ML and parser-based approaches.

E. Unified Approach

This approach fuses characteristics of the term based classification and the parser-based approaches.

1) Adding parsing results to Naïve Bayes: Naïve Bayes is not limited to use *terms* as features: One can include any relevant aspect as a feature in the classification process. Given these premises, we add the parser-based classification output to improve the Naïve Bayes ML process. We do this by adding four new features to the feature-vectors, in addition

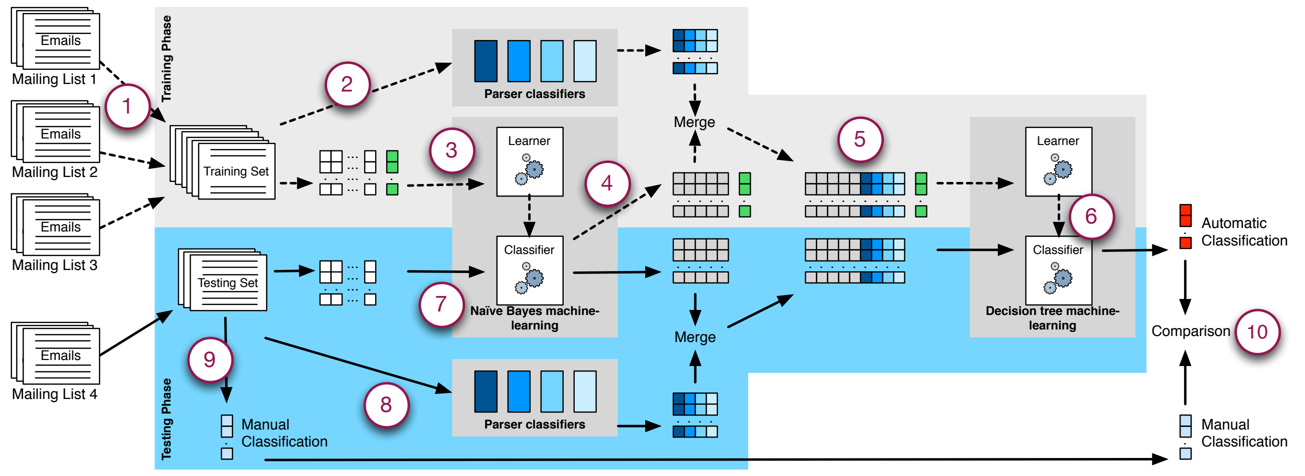


Figure 4. Training and Test Process of the Unified Classification Approach

those presented in Section V-A. Each new feature maps the output of a parser: The value is 1 when the corresponding parser matches the specific line, 0 otherwise. We used Naïve Bayes and performed mailing list cross validation.

Varying the value of the threshold t (see Section V-C), we found the best average results to be at $t = 11$. Table VI shows the confusion matrix on the best results achieved by adding the four parser-based features to the Naïve Bayes approach. It correctly classified 62,093 instances (91.3%), 1,513 more than the previous approach.

Table VI
Results adding parser-based features

classified as →		NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	■	31,898	960	97	158	29	0.908	0.962	0.934
Junk	▒	3,087	25,787	325	203	183	0.962	0.872	0.915
Patch	▧	55	29	1,719	278	1	0.739	0.826	0.780
Source Code	▦	78	13	185	1,636	2	0.719	0.855	0.781
Stack Trace	▨	9	6	1	0	1,053	0.830	0.985	0.901

Comparing the confusion matrices of the ML approaches (Table VI and IV), we see that the new features helped to decrease the instances wrongly classified as NL text. Being NL the most frequent class (see Table II), it has a strong impact on the evaluation of the MAP hypothesis of Naïve Bayes (see Section V-A1); since the new features reduced the NL class impact, they play a major role in the classification.

Although achieving the best results so far, this approach has drawbacks. First, we note that both *patch* and *code* have more than 150 wrongly classified instances: This contradicts the high precision value reached by the single parser classifiers. It is probably due to the fact that, even if these parser features have a high weight in the computation, they are at the same level of the other features that, being a large number, also influence the results. We expect an approach not having the conditional independence assumption of Naïve Bayes to better model the new features, which are highly inter-dependent. In the following we explore a two-pass classifier approach to better exploit parsers, yet relying on Naïve Bayes qualities.

2) Unified Classification Approach: To explain our unified classification approach, we refer to Figure 4. The idea behind this approach is using Naïve Bayes to evaluate a partial classification only on the features based on *terms*, and then use another ML classifier to model the fusion of Naïve Bayes results and parser-based classifications.

Training: We first (Point 1) extract the emails from the three mailing lists on which we want to train the ML algorithms, then, we provide them—along with the manual classification—both to the parser-based classifiers (Point 2) and to the Naïve Bayes learning algorithm (Point 3), in the form of feature-vector on words, punctuation, bi-grams, and context. Naïve Bayes trains a classifier, but instead of returning the instance classifications, it outputs a 5-dimension vector for every line: Each dimension represents a class (*e.g., junk*) and the value is the probability—evaluated by Naïve Bayes—that the line belongs to that class. In other words, instead of picking the highest value and providing the final classification, we output all the 5 probabilities and we map them to features, thus reducing the initial features to 5. At the same time, the parsers create other four features, as in Section V-E1. Once both feature sets are evaluated, they are merged into a vector of 9 dimensions, plus the manual classification (Point 5). This vector is treated by another ML algorithm to train the final classifier (Point 6): The actual output of the training.

The choice of the ML technique for the second step is critical: We need an algorithm to correctly model the peculiar characteristics of our features. We tried a number of different ML approaches. The decision tree [26] (broadly used in data mining) is the most suited algorithms, because it is favorable to the parsers' features, which are almost mutually exclusive. **Testing:** The test process is depicted in the bottom half of Figure 4. We take emails from the fourth mailing list and we remove the manual classification. Then, we provide the emails to the parsers (Point 8) and create the feature-vectors, to be given as an input to the previously trained Naïve Bayes classifier (Point 7). Subsequently, the output of the two technique is merged in a unified 9-dimensions vector, which it is used as input to the second ML classifier, previously trained, which outputs the final classification. We compare this classification (Point 10) to the manual one (Point 9) and we evaluate the results. The training and test phases are repeated 4 times rotating the four mailing lists.

Table VII
RESULTS OF THE UNIFIED APPROACH ON MAILING LIST CROSS VALIDATION

classified as →	NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text ■	31,584	1,470	0	87	1	0.937	0.953	0.945
Junk ▦	1,958	27,498	12	115	2	0.943	0.929	0.936
Patch ▧	68	49	1,935	30	0	0.990	0.929	0.959
Source Code ▩	86	118	8	1,702	0	0.880	0.889	0.885
Stack Trace ▨	18	12	0	0	1,039	0.997	0.972	0.984

We tested the approach with a range of t values and the highest ratio of correct instances (94.1%) at a t value of 120, which lies within the range described in Section V-C. The lowest ratio of correct instances with a t value (*i.e.,* 11) within the range is 92.1%; out of the range, values are lower.

Table VII shows the results achieved by the approach on the best t value. This two-steps approach, which differently merges and model the information, improves the results for all the classes by increasing not only the results related to the parser classifiers (*i.e.,* patch, stack trace, and code), but also those connected to the Naïve Bayes algorithm. The F-measure values are all increased, with a decrease in precision of junk classification and in recall of NL classification, probably due to the overall lower weight given to Naïve Bayes results.

VI. Threats to Validity

Construct Validity threats regard the relation between theory and observation, *i.e.,* measured variables may not measure conceptual variables.

To classify email content we rely on error-prone human judgment. To alleviate this issue, we devised a web application to ease the annotation process. Two annotators cross-inspected 10% of the emails. They found only 12 erroneously classified lines. We corrected these 12 errors in the set of email that was used for the experiments. We expect the same low error proportion in the rest of the sample, which may affect the accuracy of the results.

Statistical Conclusion threats are concerned with whether we have enough data to support our claims.

We took samples of email populations representative with a 95% confidence and a 5% error level, which are standard values. On the number of lines, our corpus has 67,792 not empty lines.

External Validity threats are concerned with the generalizability of the results.

The approaches we tried may show different results when applied to other software systems and mailing lists. To alleviate this, we chose 4 systems with unrelated characteristics and developed by separate communities. The usage of the mailing list varies, as confirmed by the different line class distributions. To test the generalizability of our approach we conducted cross mailing list validation. A second threat concerning the generalizability is that our approach is tailored to a single object-oriented programming language, *i.e.,* JAVA. However, since most of the language related line recognition relies on island parsers (see Section V-D), it can be easily adapted to other programming languages that have a similar structure (*e.g.,* C#, PYTHON), without the need of changing the ground concepts we used.

VII. Conclusion

Email communications contain valuable information to support software development, comprehension, and analysis. In this paper, we contribute a novel technique to automate the analysis of such valuable, but also voluminous, data that is specifically tailored for software engineering.

In particular, we presented a unified 2-step approach that fuses automated supervised ML approaches with island parsing to perform automatic classification of the content of development emails into five language categories: NL text, source code fragments, stack traces, code patches, and junk. The results obtained are very positive, even with cross mailing list validation. In fact, parser-based classifiers are mailing list independent and offer a solid basis made more robust by the probabilistic ML approach.

This work is a step toward a more effective exploitation of email data, for example by allowing improved traceability recovery techniques, refined artifact summarization approaches, and more precise fact extraction methods.

As a future work, we plan to investigate whether other classification techniques (such as INFOZILLA [8]) can be included in our unified approach to improve and strengthen the overall results.

All data sets of the experiment and source code can be found at the paper's companion website located at http://mucca.inf.usi.ch.

ACKNOWLEDGMENT

Bacchelli gratefully acknowledges the Swiss National Science foundation's support for the project "SOSYA" (SNF Project No. 132175) and the European Smalltalk User Group (http://www.esug.org). Dal Sasso gratefully acknowledges the support of CHOOSE, the Swiss Group for Object-Oriented Systems and Environments (http://choose.s-i.ch).

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering (TSE)*, 28(10):970–983, 2002.

[2] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci. Extracting structured data from natural language documents with island parsing. In *Proceedings of ASE 2011 (International Conference On Automated Software Engineering)*, 2011.

[3] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting source code from e-mails. In *Proceedings of ICPC 2010 (18th IEEE International Conference on Program Comprehension)*, pages 24–33. IEEE Computer Society, 2010.

[4] A. Bacchelli, M. Lanza, and V. Humpa. RTFM (Read The Factual Mails)–Augmenting program comprehension with REmail. In *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)*, pages 15–24, 2011.

[5] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 375–384. ACM, 2010.

[6] M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. Factorial hidden markov models. In *Machine Learning*, pages 29–245. MIT Press, 1997.

[7] A. L. Berger, V. J. D. Pietra, and S. A. D. Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of MSR 2008 (5th International Workshop on Mining Software Repositories)*, pages 27–30. ACM, 2008.

[9] N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proceedings of ICSM 2009 (25th International Conference on Software Maintenance)*, pages 539 –542. IEEE, 2009.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? Bias in bug-fix datasets. In *Proceedings of ESEC-FSE 2009*, pages 121–130. ACM, 2009.

[11] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in OSS projects. In *Proceedings of MSR 2007*, pages 26–29. IEEE Computer Society, 2007.

[12] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of ICML (23rd International Conference on Machine learning)*, pages 161–168. ACM, 2006.

[13] V. R. Carvalho and W. W. Cohen. Learning to extract signature and reply lines from email. In *Proceedings of CEAS 2004 (1st Conference on Email and Anti-Spam)*, 2004.

[14] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *International Journal on Empirical Software Engineering (EMSE)*, x(x):to be published, 2011.

[15] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proceedings of MSR 2004*, pages 22–26, 2004.

[16] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[17] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution*, 18:207–236, 2006.

[18] T. Gleixner. The realtime preemption patch: Pragmatic ignorance or a chance to collaborate? In *Keynote of ECRTS 2010 (22nd Euromicro Conference on Real-Time Systems)*, 2010. http://lwn.net/Articles/397422/.

[19] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of ICSE 2010*, pages 223–226. ACM, 2010.

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

[21] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 2nd edition, 2009.

[22] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of ICSE 2011*, pages 351–360, 2011.

[23] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[24] W. Lidwell, K. Holden, and J. Butler. *Universal Principles of Design*. Rockport, 2003.

[25] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[27] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[28] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of ICSE 2010*, pages 505–514. ACM, 2010.

[29] L. Renggli, S. Ducasse, Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *Proc. of DYLA 2010 (4th Workshop on Dynamic Languages)*, 2010.

[30] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE TSE*, 25:557–572, 1999.

[31] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34:1–47, 2002.

[32] O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proc. of the 12th Conf. on Computational Linguistics*, pages 636–641, 1988.

[33] J. Tang, H. Li, Y. Cao, and Z. Tang. Email data cleaning. In *Proceedings of KDD 2005 (11th ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 489–498. ACM, 2005.

[34] A. E. H. Thanh H. D. Nguyen, Bram Adams. A case study of bias in bug-fix datasets. In *Proceedings of WCRE 2010*, pages 259 –268. IEEE CS Press, 2010.

[35] M. Triola. *Elementary Statistics*. Addison-Wesley, 2006.

[36] G. Venolia. Textual allusions to artifacts in software-related repositories. In *Proceedings of MSR 2006*, pages 151–154. ACM, 2006.

[37] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE TSE*, 36(5):618–643, 2010.