Fine-grained Content Classification of Development emails

Master's Thesis submitted to the

Faculty of Informatics of the Università degli Studi di Padova

Master of Science in Informatics

Curriculum of Artificial Intelligence

presented by

Tommaso Dal Sasso

under the supervision of

Dr. Fabio Aiolli Alberto Bacchelli Prof. Dr. Michele Lanza

14 December 2012

A mio padre, che mi ha sostenuto ed incitato in ogni momento della mia formazione.

A mia madre, la cui curiosità mi ha ispirato ad essere una persona che ama imparare.

Dedico questo mio lavoro ai miei genitori, che con il loro esempio di impegno e coerenza mi hanno reso la persona che sono oggi e che sarò domani.

Abstract

Electronic communication means have become crucial with the adoption of software at any level of everyday life; playing a central role in the coordination of collaborative, distributed development. One widely used means is represented by emails from development mailing list, which contain questions and discussions among developers about design choices and various issues concerning the evolution of a system. The contents in these archives could lead to consistent information *e.g.,* describing the system's history during its evolution, or giving a meaningful insight on the currently active development tasks. The analysis of emails poses, however, some serious challenges: the text is often noisy, and usually contains characters that make it hard to build a robust parser to automatically extract accurate information. Previous research addresses these issues by working at the document level, without precisely discriminating the various sections that compose a development email.

In this thesis we present a classification approach, which integrates machine learning, information retrieval and parsing techniques, to enable robust email content analysis at the line level. The developed technique classifies email lines in five categories—*i.e.,* text, junk, code, patch, and stack trace—that allows one to subsequently apply ad hoc analysis techniques to exploit the peculiarities of each category and removing the sections of the content that would entail noise, reducing the effectiveness of the analysis. We evaluated our approach on a statistically significant set of emails gathered from mailing lists of four unrelated open source systems, reaching high values of precision and recall.

Acknowledgements

This long voyage finally gets to an end. During these years, I met many people that gave me a bit of inspiration to grow and improve myself.

First of all, I would thank the REVEAL Group for adopting me during my thesis experience: Alberto for being a mentor, a sincere friend and my best man; Michele for teaching me that fun is a central part of a creative and inspired work, and that one needs love to make a good coffee; Marco, Fernando and Lile, for the kind support and the passion they showed in their everyday work.

I would thank my parents and my family for educating me towards continuous self improvement. I would thank my wife for her patience and the smiles that let me go through all these years.

I would thank AGESCI for making me *see* a better world, and all the people I worked with to help make this dream day by day a little more concrete.

Thank to all the students that shared a bit of their path with me, particularly to the Son of a Bit Company, for teaching me about the game, and that *"An artificial intelligence algorithm is like a piece of marble, that the programmer has to sculpt to fulfill his needs"*.

Contents

Contents **viii**

List of Figures **ix**

1 Introduction **1**
1.1 Goal 1
1.2 Domain and Motivation 2
1.2.1 Problems with the current approach 3
1.2.2 Bag of words and emails analysis 4
1.3 Contribution 7
1.4 The thesis experience 8
1.5 Structure of the Document 8

2 Related Work **9**

3 Parsing and Machine Learning **13**
3.1 Parsing 13
3.2 Machine Learning 14
3.2.1 Naïve Bayes 15
3.2.2 Decision Trees 16
3.2.3 Support Vector Machines 17
3.2.4 Methods implementation 19

4 Data Collection & Classification **21**
4.1 Data collection 22
4.2 Data classification 24
4.3 Data distribution 27

5 Experiment **29**
5.1 Term based classification 30
5.1.1 Selection of the terms 31
5.1.2 Training and Evaluation Methodologies 34
5.1.3 Term Based Features and Overfitting 35
5.1.4 Results 36
5.2 Parsing based classification 38
5.2.1 Stack trace parsing 39
5.2.2 Patch parsing 40
5.2.3 Source code parsing 41
5.2.4 Junk parsing 42
5.2.5 Results 44
5.3 Mixed Approach 46
5.3.1 Adding parsing results to Naïve Bayes 46
5.3.2 Results 47
5.3.3 Unified Classification Approach 48
5.3.4 Results 50

6 Discussion And Threats to Validity **51**
6.1 Discussion 51
6.2 Threats to Validity 53
6.2.1 Construct Validity 53
6.2.2 Statistical Conclusion 53
6.2.3 External Validity 53

7 Conclusion **55**
7.1 Contribution 56
7.2 Exploitation 57
7.3 Future work 58
7.4 My contribution to the experiment 58
7.5 Epilogue 59

Bibliography **61**

Figures

1.1 Example development email with mixed content 5

3.1 The new instance (semitransparent) is near to correctly classified training data, but is incorrectly classified 18
3.2 The SVM maximizes the *margin* 19
3.3 A dataset containing data that are not *linearly separable*: there is no 1-dimensional line that separates the black instances from the white one. 20

4.1 Mailpeek: our web app for classifying email content 25

5.1 Lines modeled as vector of terms. Each element is a feature that contains the frequency of the term in the line. 30
5.2 Results on training and test sets, by line threshold for features . . 36
5.3 Training and Test Process of the Unified Classification Approach . 45

Chapter 1

Introduction

1.1 Goal

Last decade showed a massive adoption of software systems in many aspects of human activities, causing a great number of software project to be developed. Such projects usually grow over time in number of features and in lines of code, and employ an increasing number of people working on the development and mantainance process; these people need to coordinate their efforts, and doing so they generates great volumes of communication.

This communication is sometimes stored with the software repositories of the project they refer to, and represent an archive of information that describes the *history* of the system during its growth. Being able to access this history can provide useful information for supporting software analysis and program comprehension tasks [19]. By analyzing different repositories one has access to different perspectives on systems; for example, issue repositories open a view on defective entities, thus enabling research on defect location and prediction [16]; similarly, natural language (NL) documents, such as emails, provide a view on developers' interactions and opinions [34].

Obtaining benefits from the history of a system—to support program com-

prehension and software development—is not trivial: the information extracted from software repositories is not always structured and follows variable conventions derived, for example, from spoken language or from coding style. However, to perform a reliable analysis we need information that is *relevant*, *unbiased*, and *comprehensible*. In this vein, researchers are analyzing the quality of issue data sets, to verify if approaches are accurate enough to provide flawless information [11, 38], and to determine what information is more relevant, for example in bug reports [40] or among the changes in version histories [26].

A clear example of communication related to a software system comes from development mailing lists that developers use to report bugs, announce version releases, discuss new features and receive feedback from the users. Development emails contain different sections like natural language, code, signatures or quoted phrases, and does not refer to a defined and consistent structure. A crucial point to consider when we want to extract information from these unstructured or semi-structured documents, where natural language may be a consistent part of the whole corpus (*e.g.,* web pages, IRC chat logs, emails from mailing lists), is the quality of the data we process. In fact, in natural language we cannot rely on a standard and unique grammar, because it is subject to irregular changes—for example a neologism or an error—as opposed to standard structures of programming language keywords or bug reports that are checked by a parser. Moreover, such documents often contain additional automatically generated content, like replies and indentation symbols in emails, tags in html pages, or time and author information in IRC chats, which are usually considered as noise during the analysis of the content. The risks of using email data without a proper pre-processing phase that filters noise and unstructured text, is presented by Bettenburg *et al.* [9], providing evidence with examples that show how noise severely impacts such data.

1.2 Domain and Motivation

We believe that the problem that we tackled in this work represents a concrete need with practical applications. We show that the current state of research has not yet explored this area, and that fine-grained email content classification can

help improving the performance of software repositories analysis.

1.2.1 Problems with the current approach

Even when an accurate data cleaning is conducted, the analysis performed by current techniques treats each document as a single *bag of words*: The document is stored as a list of terms found in the document with frequencies associated to each word. This simplification comes from the field of information retrieval, which usually deals with well-formed natural language documents written by information professionals, such as journalists lawyers, and doctors [29]. This representation allows only a coarse-grained analysis, since it gives no inspection inside the document structure. This classification can assign an entire document under a single category, thus failing to separate and classify the different sections in the content. As a result, the information that we can extract with this technique is whether a document belongs to a category or, at most, whether a document contains elements of each category. We cannot know the portion of the text belonging to class, or if a document belongs to a category for a single line or for the whole text, so, we have neither *quantitative* nor *qualitative* measures of the categories.

Applied to software engineering problems, the bag of words approach has shown to be effective in some tasks (*e.g.,* traceability between documents and code [1]), but generally performs poorly, due to a considerable amount of noise in the data that causes a reduction of the quality, reliability and comprehensibility of the available information. This degradation happens because natural language is not defined by a strict, formal grammar; in fact, it can be arbitrarily formatted, and interleaved or embedded in sections containing text of other classes (commented code).

We can also note that the repositories we want to fetch contain data that belong to a different domain with respect to standard information retrieval tasks. Each class is characterized by specific characteristic that can be exploited to help the classification process: source code contains language keywords in a strictly defined order, stack traces contain debugging information, signatures contain standard markers like "`wrote`" or the character "`>`" for indentation, while natural

language shows none of these conventions. With this additional information we assume that a more precise classification than a *bag of words* approach can be succesfully performed. Having a row level precision allows us to remove the lines that contain noise—like signatures—that would influence our analysis, thus improving the result of successive inspections. We can also consider only the lines of selected categories and filter out the uninteresting ones, to operate on a controlled subset of the original document.

1.2.2 Bag of words and emails analysis

Figure 1.1 shows the content of an example development email. Given the diversity of languages used in the example email, if we consider its content as a single bag of words, we would obtain a motley set of flattened terms without a clear context, and we would severely reduce the quality and the amount of available information. If we can automatically distinguish the parts that form such an email, we would provide better support for many tasks, as for example:

Traceability recovery. In Figure 1.1, 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. 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.,* [3]), would not be able to recognize the artifacts appearing in NL context and increase the relevance of the link. Such a method can only use the number of occurrences to weigh more certain terms [29], leading to imprecise results. In Figure 1.1, a weighting based on occurrences would give the most relevance to class *MPackage* (mentioned 5 times), which is marginal to the discussion. By recognizing the context in which a term appears, we can improve the quality of traceability links, by providing additional information to the user.

Common terms removal. To better characterize documents, information retrieval research suggests to remove *stop words* (*i.e.,* very common words) [29], thus giving more weight to those terms that are peculiar of a document. This

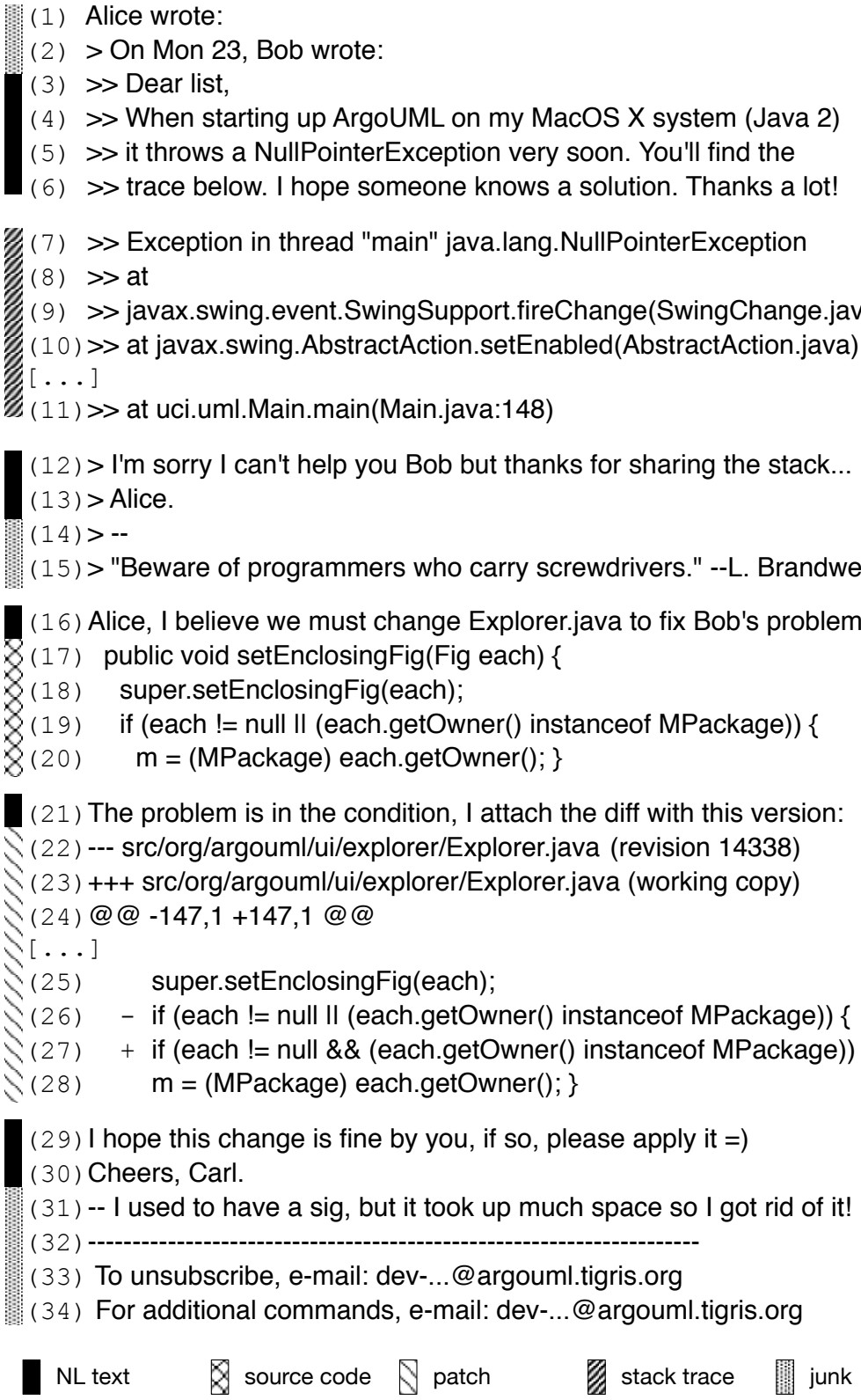

Figure 1.1: Example development email with mixed content

approach applied to development emails does not bring the expected benefits: By removing stop words, we do reduce the noise in NL parts, but we also delete information in parts that do not share the same vocabulary (*e.g.,* source code); similarly, we delete important information by removing programming language keywords from NL. For example, the removal of the stop word "each" from the whole content of Figure 1.1 would imply the deletion of a variable name in both a code fragment (lines 17–20) and a patch (lines 25–28). This is suboptimal, since variable names are known to provide relevant information [27]. By recognizing the different parts that compose an email, we can adopt different common terms removal techniques, enabling us to expose the most relevant information.

Artifact summarization. Given the amount of data produced during a system's evolution, researchers investigated how to expose the significant details to reduce information overload (*e.g.,* [32]). Techniques devised so far are tailored to specific types of artifacts (*e.g.,* source code [21], NL documents [24]) and results might be poor when applied to mixed documents, such as development emails. By recognizing the different parts that compose an email, we can devise an approach that applies the most suited summarization technique according to the type of each part and extract the correct keywords.

Content parsing. To know the exact meaning of code fragments, patches, or stack traces, we can use ad hoc parsers; similarly, NL text can be effectively analyzed with NL processing techniques [25]. For example, in Figure 1.1, using a parser for patches, we can recognize that the file being modified is *Example* (from lines 22 and 23), thus extracting further information on which issue is discussed. 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 1.1, 8 lines (marked as "junk") out of 34 contain irrelevant data. Previous research indicated how some changes in version history are not essential, and how their detection and filtering can improve change-based analysis techniques [26]. On the same line, we argue that the detection and removal of junk from email contents increase the quality of the data, and thus improve analyses based on such data. By recognizing the noise in emails, the important data emerges, improving the information

extraction quality.

1.3 Contribution

We present an approach, based on a combination of parsing techniques and machine-learning methods, that we used to classify the contents of development emails in five categories:

1. natural language
2. source code
3. patch
4. stack trace
5. junk (text with no valuable information, such as auto-generated authors' signatures or disclaimers)

The developed technique works at the line level. We created a web application to manually classify email content in the chosen categories. We classified a statistically significant set of emails taken from four open source JAVA systems, which we then used to evaluate the accuracy of our approach, obtaining promising results.

The contributions of this thesis work are:

1. a novel approach, which combines parsing techniques and machine-learning methods, for fine-grained classification of email contents;
2. a web application to manually classify email contents;
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,
4. the empirical evaluation of our approach on the benchmark.

1.4 The thesis experience

This thesis is a result of the work performed during a six month internship at the REVEAL group at Università della Svizzera Italiana in Lugano, occurred from October 2010 to April 2011. REVEAL is a research group under the guidance of professor Michele Lanza, and is active in Software Engineering, Evolution and Visualization. In particular, our work involved supporting the team in automatically elaborating emails to extract useful information from development mailing lists. The work produced a conference paper that was published at ICSE 2012 [5]

1.5 Structure of the Document

In Chapter 2, we describe our contributions with respect to related work. In Chapter 3, we present the parsing techniques adopted during the development of the experiment and the machine learning algorithms used to complement the parsing results. In Chapter 4 I show how we collected and manually annotated the data to test the new approach. In Chapter 5, we detail the classification methods and describe the evaluation. The evaluation results and threats to validity are discussed in section Chapter 6. In section Chapter 7, we conclude by summarizing the contribution of our work and by evaluating my thesis experience.

Chapter 2

Related Work

Numerous researchers applied natural language analysis techniques to software related documents, also proposing approaches to improve the comprehension of the NL parts. For example, Dekhtyar *et al.* [17] discussed the opportunities and challenges for text mining applied to software artifacts written in NL. In the following the focus is on research dealing with the recognition of the different parts that compose NL artifacts.

As previously mentioned, Bettenburg *et al.* presented the importance of a proper cleaning pre-processing phase before using email content [9]. Their idea revolves around noise in the email data, and they suggest a number of filtering heuristics to recognize noise and irrelevant information. The focus of their work is more on making the research community aware of the email data problem, than on validating the proposed approaches.

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]. InfoZilla is composed of four independent filters, one per category, which are used in cascade to process the text. The source code filter uses a parsing approach based on text matching implemented through regular expressions. The authors reported results on the effectiveness of infoZilla in *differentiating* documents, *i.e.,* deciding whether they contain or not each category. They reached almost perfect results, with values

well above 0.95 for both precision and recall in all the categories. Subsequently, they effectively used infoZilla in a practical application, to investigate which features are relevant when submitting bug reports [40].

Compared to bug comments, development emails contain some substantial differences, that present the following issues:

1. they contain a larger NL vocabulary, since the discussion is not limited to bug related issues, the content of emails can treat topics that range from deadlines to everyday life or politics;
2. they present more noise, generated for example by email headers and authors' signatures, or consistent quote sections in replies;
3. emails pose bigger challenges in text recognition, since many email clients automatically wrap long lines of text, thus breaking the right formatting [10].

Bird *et al.* proposed an approach to measure the acceptance rate of patches submitted via email in open source projects [10]. They extracted code patches from emails and used them to analyze the developers' interactions. Since the analysis was their main focus, the authors provided little information about the extraction technique and the data used to assess it.

A number of information retrieval approaches targeted the classification of text or the recognition of information with specific patterns [25], especially by exploiting probabilistic and machine-learning models (*e.g.,* Maximum Entropy Models [7] or Hidden Markov Models [6]). In particular, Tang *et al.* addressed the issue of cleaning the email data for subsequent text mining [37]. The authors proposed a cascaded approach to clean emails in four passes:

1. non-NL text filtering,
2. paragraph recognition,
3. sentence boundaries detection,
4. word normalization.

In the first pass, the method 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 non-software related emails [14]. They worked at the line level and tested the effectiveness of a set of features with many machine learning classifiers. In the signature detection task, the methods reached an f-measure value of 0.97.

Summing up, previous approaches differ from the one we present in this work as they:

- addressed more compact classification tasks, for example only detecting patches [10] or signatures [14];
- 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 [2, 37]).

During the development of the method, we strove for an approach with a wide breadth and fine granularity, able to provide a row-level detail of information for increasing the quality of subsequent analyses that could give a more accessible and useful description of the information contained in development mailing lists.

Chapter 3

Parsing and Machine Learning

We present the main methods that we used during the experiment. These approaches have proved to be effective when applied to a data mining application. We applied them to our problem to verify their effectiveness in the context of email content classification.

3.1 Parsing

Parsing is a technique that performs the syntactic analysis of a text identifying a sequence of tokens that satisfy a set of rules, called a *grammar*. A *parser* is an algorithm that takes some data in input (for example a fragment of text) and checks it against the rules of the grammar. If the input is syntactically correct, it is accepted, otherwise it is rejected.

For example, we can imagine a parser as a function that searches a text for all the occurrences of email addresses. The algorithm then will search a **username** followed by the character `@` (at) and the **domain**, where:

username is a **word**

word is a sequence of alphanumerical characters without spacing

domain is a sequence of two or more **word** separated by the character *dot*

With this definition we can easily construct the grammar of the parser:

```
email := username@domain
word := character
username := word
domain := word(.word)+
```

If the parser analyses a text like *"To submit bugs please send an email to* `tommaso@studenti.unipd.it`*"*, it will recognize `tommaso@studenti.unipd.it` as accepted by the grammar and ignore the other parts.

To refine our results, we can further restrict the number of valid addresses specifying a set of valid top level domains, like `.com`, `.it` or `.org` to get more precise results.

Using a parser allows us to specify in detail the rules that define what is accepted by the parser and what is not. We can continue adding constraints and get increasingly refined results that can be modified interactively during the analysis of a new domain. Also, the knowledge produced and contained in the grammar is easily accessible and can be extracted and interpreted.

The use of a parser has some drawbacks. First of all, it requires that the grammar is defined by an expert that has prior knowledge of the domain and can formalize his expertise into a set of rules. Such rules need to be directly derived by the information we have on the data. Also, the generated parser is specialized for a given domain, and it usually needs to be reconfigured or rewritten when applied to a new one.

3.2 Machine Learning

Machine learning is concerned with building programs that evolve during time with the experience gathered from the data. It generally consists of two steps:

a learning, or training, phase and a classification phase. During the learning process, the algorithm takes the input data and tries to build a *classifier* by inductively extracting knowledge from the examples. Such a classifier is then used in the classification phase, to make predictions on new instances of the dataset.

Machine learning is an approximation task. This means that it usually perform worse than a pure algorithmic approach. However, it is of great value when dealing with certain kinds of problems, like when the domain is too complex and impossible to completely define. For example, given its ability to generalize its experience finding patterns in the training examples, a machine learning approach can bring substantial improvements when the input data is too big to be manually inspected, or contains errors that would make a parsing approach much harder and less reliable.

3.2.1 Naïve Bayes

Naïve Bayes is a method of *supervised* learning, *i.e.,* a category of machine-learning algorithms that uses classified training examples to infer the classification function. Naïve Bayes relies on the assumption that the presence of a feature is unrelated to the occurrence of the other features. Even though this *conditional independence* assumption is a strong simplification, the method has often been found to outperform more sophisticated techniques [25]; in particular, in the text classification task, researchers showed that the simple probabilistic classifier *Naïve Bayes* achieves significant results [13].

The method uses the Bayes rule [25] to compute the probability that a document d, composed of t_k terms, belongs to a class c:

$$P(c|d) \propto P(c) \prod_k P(t_k|c) \tag{3.1}$$

To classify an instance we calculate the posterior probability $P(c_i|d)$ for each considered class, and select the ones with the highest probability. This is called *maximum a posteriori* (MAP) hypothesis:

$$C_{MAP} = \arg\max_{c_j \in C} P(c|d) \propto \arg\max_{c_j \in C} P(c) \prod_k P(t_k|c) \tag{3.2}$$

If, for example, we want to classify the instance $d =$ "$Alice\ wrote:$" as $text$, $junk$, or $code$, the algorithm first computes the probabilities as: $P(text|d) = 0.43$, $P(junk|d) = 0.55$ and $P(code|d) = 0.02$, then selects the value 0.55, and finally classifies d as $junk$.

Given the high number of probability multiplication performed, the calculated values may become too small to be represented by float numbers: This may introduce the risk of underflow. To avoid this issue, Naïve Bayes computes the values as logarithms. Moreover, when a term does not occur in the training test, the calculated probability would be zero, thus Naïve Bayes also applies a Laplacian smoothing to the product. 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.

3.2.2 Decision Trees

Decision trees are a kind of supervised learning algorithms to approximate a classification function when its values are discrete. They are one of the most used methods in practical application for their efficiency and the properties of the generated classifier.

The output of the algorithm is a tree that represents the learned function. In the decision tree, each attribute or feature is mapped to a node of the tree and each branch that starts from the node represents one of the possible values of the attribute. The classification is then performed starting from the root of the tree and descending it choosing a value for each node. When we reach a leaf, the path from the root to the leaf contains the values of the attributes of the instance, with its classification.

Various strategies consist in different approaches to decide the order of the attributes, sorted starting from the root; one of the most common methods uses

entropy as a measure to select this sorting. Entropy measures the purity of an arbitrary group of elements, *i.e.,* how homogeneous are the elements in a set. To select the ordering of the elements during the training phase, the algorithm chooses the attribute that best separates the instances, which is the one that maximizes the reduction of the entropy; the difference of entropy is called *information gain*.

One of the biggest advantages of decision trees is that a tree can be completely converted into a set of logic rules that describes its behavior. As such, the experience gathered from a decision tree is easily comprehensible by a human, and so it is reusable.

3.2.3 Support Vector Machines

Linear classifier

SVM is a machine learning method to perform supervised learning. It founds on the idea that a classifier on a p-dimensional space can be found considering a $(p-1)$-dimensional *hyperplane* that separates the instances. Its functioning resembles another popular machine algorithm, the *perceptron* [31], but the two methods differ as they use a different approach in the selection of the hyperplane chosen as classifier. The perceptron algorithm is a simple supervised learning algorithm that starts from a random hyperplane and then iteratively aims to reduce the error on the training data until it finds a hyperplane that correctly classifies all the training instances. This approach may lead to errors when new data is classified, since instances near correctly classified examples may be misclassified, as shown in Figure 3.1

Support vector machines select instead the classifier that maximizes the *margin* [12], which is the distance from the nearest instance of training data, from both the sides of the classification, as shown in Figure 3.2. This assumption tries to overcome the overfitting problem of the perceptron shown above.

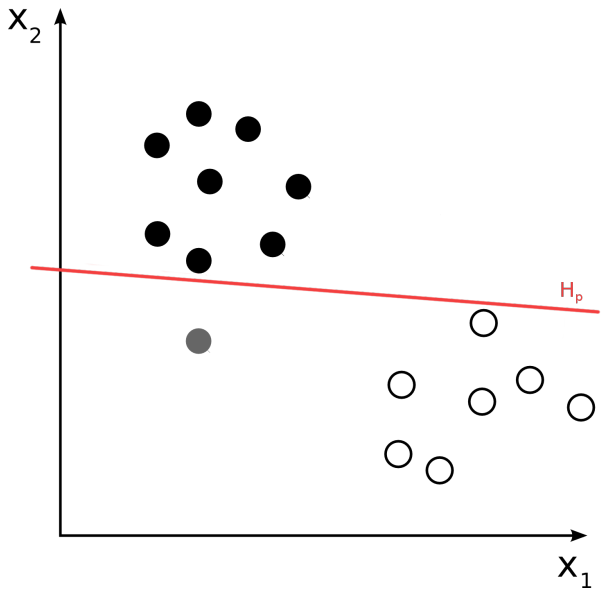

Figure 3.1: The new instance (semitransparent) is near to correctly classified training data, but is incorrectly classified

Nonlinear classification

As the perceptron, Support Vector Machines with a *linear kernel* can only train classifiers that perform a *linear separation* [23]. This means that the classifier relies on the assumption that the data instances are perfectly separable with a hyperplane, as the example in Figure 3.3. Incorrect values can be considered as *penalties* that are added to the margin, to minimize the number of tolerated incorrect classification, but still produces a sub-optimal classification. To address this issue, researchers have introduced the use of *nonlinear kernels*. A kernel is a function that maps a high dimensional space into a smaller one, allowing easily representation of functions such as inner product, and thus permitting computation that would otherwise be too complex to handle. With the use of the correct kernel function, a SVM can classify datasets that are not linearly separable.

Since learning process is performed by modifying the coefficients associated to the vectors, support vector machines don't allow a human reading their internal representation like a decision tree, so the generated model is not easily

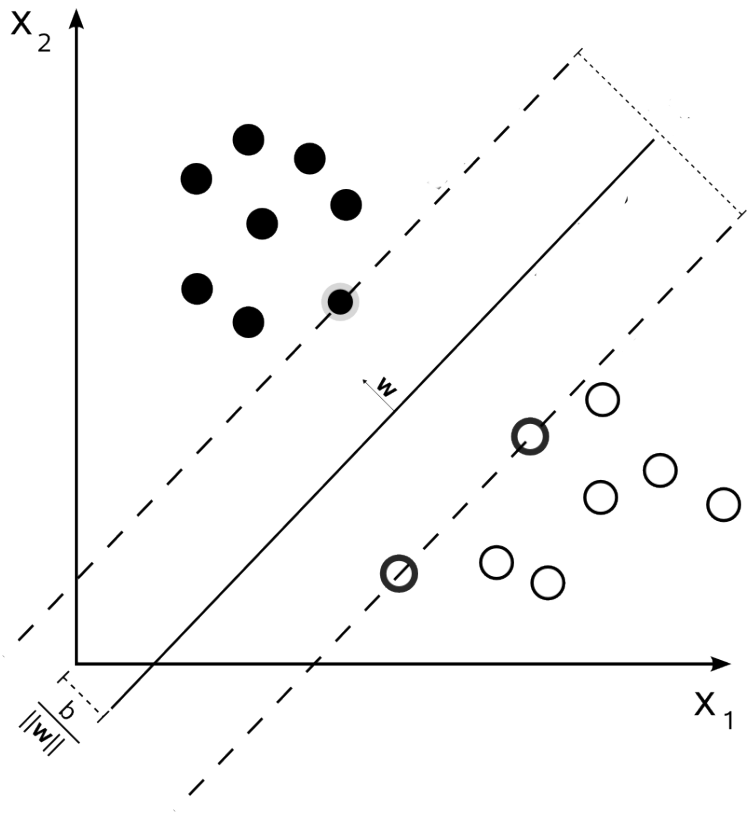

Figure 3.2: The SVM maximizes the *margin*

accessible. However, their power and relative simplicity (compared to more complex methods, such as *neural networks* [31]) made support vector machines a very effective method in data mining and information retrieval.

3.2.4 Methods implementation

To test the results of the methods we had chosen, and to quickly check the performances of possible other approaches, we used *Weka* [22]. Weka includes several state of the art machine learning and data mining algorithms. [1] Using Weka permitted us to conduct a quick evaluation phase, that showed us whether

[1] *SpagoBI*, an open source business intelligence framework, uses Weka as its data mining engine — http://www.spagoworld.org/xwiki/bin/view/SpagoBI/

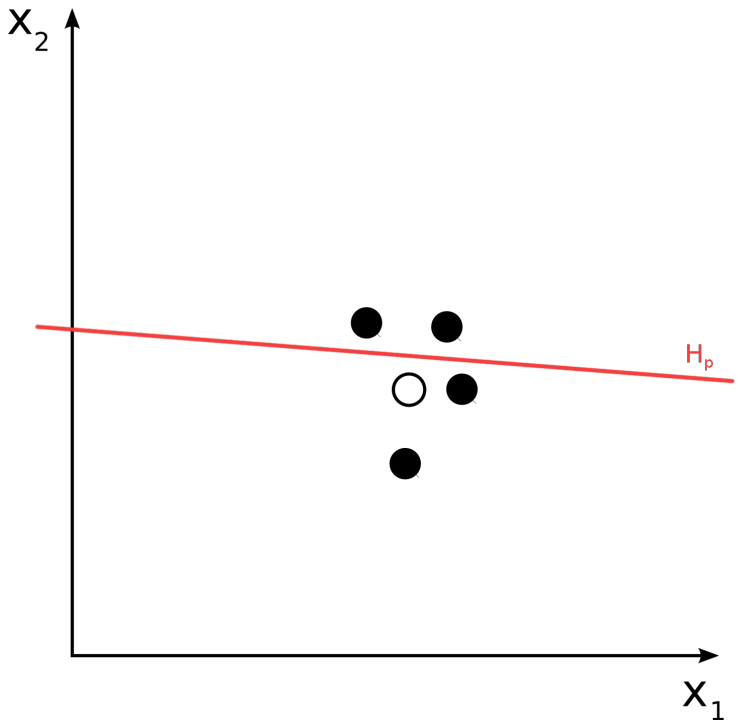

Figure 3.3: A dataset containing data that are not *linearly separable*: there is no 1-dimensional line that separates the black instances from the white one.

the the methods we had decided to use could fit the problem.

Chapter 4

Data Collection & Classification

The goal was to devise a method for reliably and precisely classifying emails at line level precision, with the aim of improving data quality and comprehension. The first step was to obtain datasets that were *accurate*, *comprehensive*, and of *statistically significant* sizes, to test the approaches we wanted to apply. These features were critical for validation purposes and to lead to more reliable training for the supervised classification methods we employ.

We implemented two tools to assist the classification process: a mailing list importer, to fetch mailing lists repositories mining which would download, format and store data, and a web application to present the gathered data and provide a practical method to assist the manual classification of email content in categories.

We integrated these tools in *Miler*, a previously developed toolset for exploring email data [4], and we called the new application *Miler2*. We then used these tools to collect the necessary data, perform some data cleaning and classify the instances to build the dataset.

4.1 Data collection

Some mailing lists provide their content in the MBOX format, which is a standard format to archive emails that stores the email content in plain text.

MBOX format was then a good choice in terms of data representation, but presented the problem that MBOX archives were not centralized, but only available through the project's website. To fetch the data we would need to visit each website, find the location of the email archive, and download it. This process would be not automatically reproducible for new projects.

We found that various distinct software applications were available for managing and consulting mailing list archives. Consequently, different software systems often use different applications to manage email repositories, and even a single system can change the mailing list interface during its lifetime. This resulted in the necessity of writing one custom importer per system and, more important, in having data that might be not consistently formatted (*e.g.,* with different line breaks). We tackled this issue by using MarkMail[1], an online service for searching among more than 8,000 up-to-date mailing lists, presented through a web application. We had access to the previous implementation of Miler, a web crawler to extract emails from MarkMail website [4], but due to structural changes to the HTML page of the emails it was not a reliable source. Because of this, and due to the need of gathering more data from the website than the previous application, we wrote a new crawler. To avoid being prone to structural changes in the web page of the service, we contacted MarkMail asking for an application programming interface or a standard procedure to gather information. The administrators provided us with a url template[2] to gather one email message in XML format, given the message id. They also specified that this is the method used by the Google[3] crawlers to fetch their emails. This format not only offered email content separated from its metadata, as the web version does too, but also provides faster access, since it does not download the unrelated data of the HTML version, and it is independent from the website formatting. The new implementation then takes advantage of these features to build a more

[1] `http://markmail.org`

[2] http://markmail.org/message.xqy?id=<*messageId*>

[3] http://www.google.com

robust parser, and uses a *thread pool* to parallelize the fetching, resulting in a faster crawler.

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

Table 4.1: Email datasets used in the experiment, by system

Table 4.1 shows the four software systems and mailing lists considered in the experiment. The selected software systems are unrelated and emerging from the context of different free software communities, *i.e.,* Apache, ArgoUML, and Freenet. The development environment, the usage of the mailing lists, and the development paradigms are likely to differ among the systems, thus mitigating the impact of the threats to external validity of experiments.

We focused on *development* mailing lists, since they contain the highest density of information related to software development, including the part of the community involved in writing and debugging code. We imported all the messages starting from the mailing list inception (second column in Table 4.1) to the end of November 2010. We filtered out messages automatically generated by the bug tracking system and the versioning system, since they do not contain valuable information related to the development of the project. Also, we extended the tool to automatically perform the filtering process making it then reproducible on any new dataset.

From each filtered mailing list, we extracted statistically significant sample sets (last column, Table 4.1). Since we had no prior knowledge available on

the distribution of line categories in the populations of the mailing lists, I opted for simple random sampling [39] to pick the emails. The chosen sizes have a confidence level of 95% and an error of 5%[4].

4.2 Data classification

To develop a supervised machine learning approach and to train and test the classifiers, we needed some pre-classified instances to use as golden set. This meant *manually* classifying the 1,493 emails in the sample datasets. Such a process is both time consuming and consistently error-prone, so we developed a module of MILER2 called MAILPEEK, a web application written in Smalltalk, using the Seaside framework [18] to help the annotation process and allow an easier and immediate visual verification of the classified instances. Figure 4.1 show the classification interface of MAILPEEK.

MAILPEEK was developed using the MVC pattern, to separate the visualization, the model and the controller layers. The Model layer uses a POSTGRESQL database to read and store the mailing list data, that the application can access through a Smalltalk module called METADB [15]. MetaDB is a database abstraction layer with support for persistency that allows one to interact with the database entries as if they were object instances. The email content is stored as text, and its classification is stored as a set of intervals referring to the position in the text. The controller layer contains the method to receive the text fetched from the database and its classification, then it combines them into a data structure that pairs each character to its classification. The presentation layer models each page as a class. When the web browser asks for a page, the SEASIDE server sends a message to the relative class, invoking the method that retrieves the data and composes the page to be rendered.

MAILPEEK presents an interface that allows the user to navigate through the gathered mailing lists. It offers two modes: a *view mode* and a *classification mode*. The view mode shows all the emails of a mailing list sorted them by date and topic. The user can inspect the emails, read their content and also

[4] see [3, 39] for more information about sample size determination.

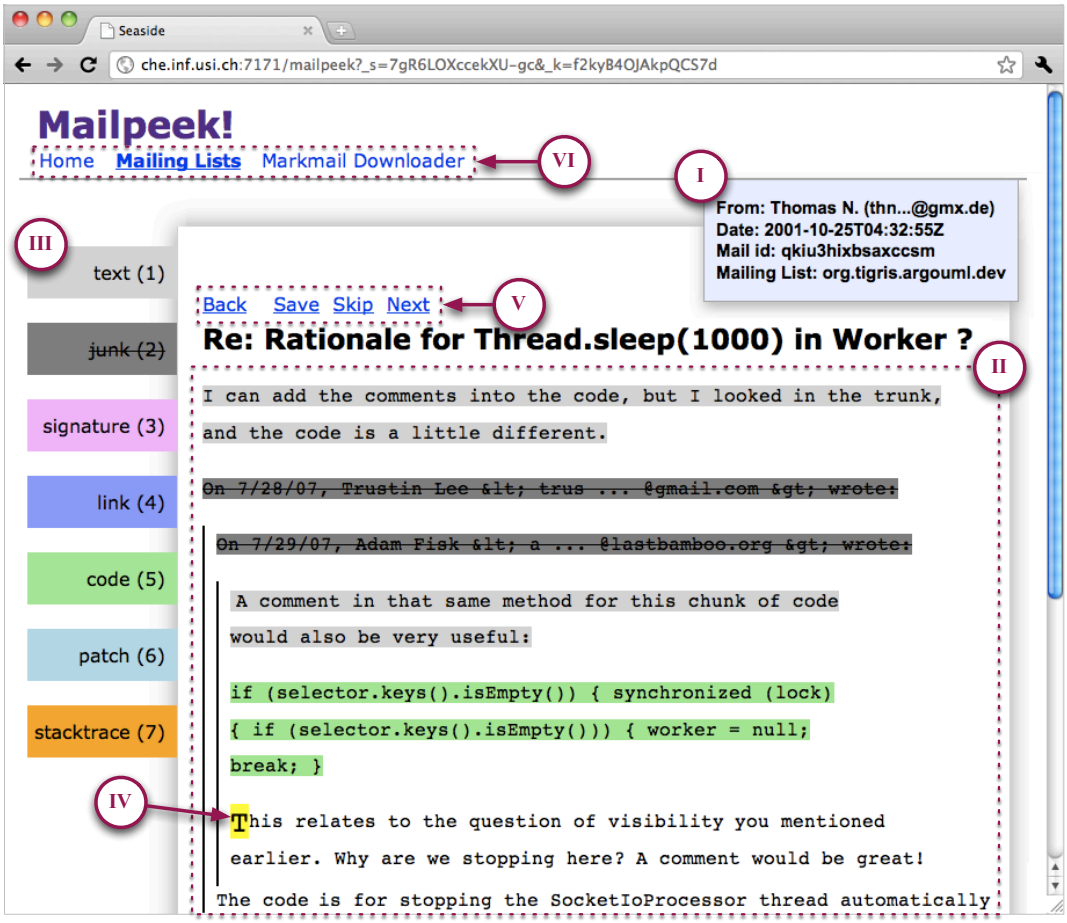

Figure 4.1: Mailpeek: our web app for classifying email content

view their properties (*e.g.,* the current classification or if the email has already been classified), to allow an easy review of the work and reduce the errors. The classification mode picks a random email among the unclassified ones from a selected mailing list excluding the emails that were previously filtered. It then opens its content presenting the classification interface to the user.

The classification interface presents the email and its metadata using a sheet metaphor. The user can select the text to classify and then choose among the categories presented at the side of the sheet as post-it. Figure 4.1 shows the classification interface 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.

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* granularity: 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.

As an important feature, when users hover with the mouse on any character in the email content area (point II), its font size triples (point IV). According to Fitts' Law [28], this eases the selection and reduces the necessary time, thus decreasing fatigue and errors.

The top menu (point VI) gives the users the ability to control the workflow with four options: *back*, *save*, *skip* and *next*. Once an email is completely classified, the user clicks on *save* to save the current classification and proceed with the next email. Mailpeek then automatically 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 *next* option goes to the next email without changing the status of the current one. This option is useful when dealing with emails that are hard to classify and need further analysis. The *back* link allows one to return to the mailing lists menu to change mailing list or trigger the MarkMail importer to download new data.

We performed the manual classification task with the help of a PhD student of the REVEAL Research Group. We conducted the analysis in two passes. First, we classified different sets of emails, then, each of us reviewed 5% of the emails analyzed by the other person. In the reviewing phase, we found only 12 erroneous lines (less than 0.2%).

	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	

Table 4.2: Distribution of the categories per line, by system

4.3 Data distribution

Table 4.2 reports the distribution of the different categories in the considered sample sets. Since the machine-learning techniques that we used are highly influenced by the distribution of categories, the data had to be carefully analyzed to achieve optimized and reliable results (see Chapter 5).

We can see that the amount of junk is critical: More than 30% of lines in any system are classified as junk. This highlights again the impact of noise on email data. The rest of the lines are mostly composed of natural language, while the frequency and ranking of other categories is lower and changes according to the mailing list (*e.g.,* Mina has a higher amount of patches compared to Freenet). The different composition of the content of these email sets reflects the different usage of mailing lists among diverse communities, thus stressing the importance of verifying our results on more than one system.

Hybrid Lines. A row level granularity allows a high degree of precision, but of course it still is an approximation. Some lines happened to be split into two sections, each one belonging to a different category, while the row level approach allowed us to consider a line as an atomic entity. As a result, we needed to treat these lines as a special case, either considering it belonging to both categories, or choosing only one classification. By investigating these cases, we found that these lines are mostly composed of a junk part not well separated by the NL text, and they are not meaningful. These lines constitute less than

5% of the population (*i.e.,* 3,362 lines). Despite the small number and their low information value, the bias in the results was reduced by still considering them as separated instances.

Chapter 5

Experiment

To implement the email classification approach, we decided to combine some ideas gathered from both the Information Retrieval and Machine Learning fields, to devise a new technique. Since the domain of the project was different. We had to adapt the existing techniques and interpret them to fit the properties of the given dataset.

During the previous case studies of the task, the research group had implemented a solution [4] that we had the opportunity to examine and integrate. This solution was based on parsing techniques (see Chapter 3), and it consisted in a battery of parsers, each one configured and tuned to detect whether a line could belong to a single category. However, given the strict nature of the parsers' rules and the noisy constitution of the emails content, such method could not overcome some limitations that bounded its performances. Our approach consisted in integrating this technique using machine learning trying to defeat the implicit limitation of the parsing approach and increment the results. Both the original approach and the new one can be used as stand alone methods, but they achieve the best results when combined, creating a *unified* approach.

5.1 Term based classification

"Most current IR systems are based on a kind of extreme version of compositional semantics in which the meaning of a document resides solely in the set of words it contains" [25]. When dealing with text, documents are usually considered as *bags of words*, where syntactic information, ordering and constituency of the words play no role in determining their meaning. In practice, IR systems model each document as a vector of features, which correspond to the *terms* that appear 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 in association with supervised machine-learning algorithms to achieve very effective results in automatic text classification [29, 35]. We decided to model the representation of the documents on the same basis: by considering the lines to be classified as **vectors of terms**, as shown in Figure 5.1.

	t_1	t_2	t_3	t_4	...	t_C
L_1	0	1	3	0	...	0
L_2	0	0	1	0	...	2
...			...			
L_N	0	0	1	2	...	1

Figure 5.1: Lines modeled as vector of terms. Each element is a feature that contains the frequency of the term in the line.

The methods we intended to use were mainly funded on statistical principles; this representation then permitted to easily store the lines of the text together with their term frequency, thus allowing me to use this information while applying machine learning algorithms for their classification.

The two main critical steps in the elaboration of the classification method

are the choice of the machine learning techniques and the representation of the vector of features. The selection of the machine learning method was crucial since it would define the way the data would be processed and used to detect patterns to build the classifier. The choice and representation of the vector of features is essential because it defines the *bias* and the part of the data that is used during the analysis; it also affects performances and it is probably the aspect that has the biggest impact on the final results.

This important decision cannot rely uniquely on results from IR field. The focus on IR methods mainly refer to other domains and classification tasks as finding relevant documents for a topic(*e.g.,* a search engine), or categorization of documents of natural language [30]. Since the selection of the applied techniques is not trivial, we describe and motivate each decision taken.

5.1.1 Selection of the terms

Words

Words are the fundamental tokens of all the languages used in the categories to be classified. We judged the words in our corpus of 67,792 non-empty lines to be valid features for a representative line modeling. In fact, the features from randomly sampled development emails, contained a variegate set of terms which commonly appeared during the threads, generating a core of high-frequency terms together with many low-frequency ones. A raw approach would consider all the words of the corpus as features, thus ending up with sparse feature-vectors of the same dimension as the corpus cardinality. On the contrary, before mapping the words, IR methods usually include a pre-processing phase to remove words that are *not* significant. This phase usually includes the following steps:

stop word removal is a process that searches the documents for very common words, and excludes them. This step is useful to remove the terms that are not informative, since they are very common and tend to flatten the differences between documents. The list of *stop words* can either come from

a precompiled dictionary or can be derived from the analyzed documents, by choosing a frequency as threshold and exclude all the terms that exceed the threshold.

stemming tries to generalize the terms found, by collapsing the morphological variants of a word into a single *stem* (*e.g.,* "model" and "models" are reduced to the root "model"). The purpose of this operation is to avoid that two variations of the same term are considered as two different tokens, thus increasing the recall and reducing overfitting. Stemming should lead to a better clusterization of similar documents.

threshold to term frequency behaves like the stop word removal, setting a threshold to the frequency of a term, but instead of excluding the tokens with the highest frequency, it sets a lower bound under which a term is ignored. This generally leads to better results by removing very specific terms such as hapaxes that hardly indicate similarities between documents. Removing these words reduces some of the noise in the data and reduces overfitting.

Through an analysis of the terms, we evaluated which of the common IR preprocessing steps could fit the task. We decided to *not* perform stop words removal, since we expected the words that appear very frequently to be representative and valuable features to describe a class. For example, stop words are likely to appear more often in natural language, while JAVA keywords in source code. A word like "if" is expected to have a high frequency inside code, and a low frequency in junk. This is clearly very informative and helpful for the classification task. We also decided to not conduct stemming, since we expected some morphological variants to be more characteristic of certain classes of lines (*e.g.,* verb tenses are widely used in natural language text, while terms like "class" in code do not have the same meaning as "classes" or "classified"). On the contrary, words with a low term frequency (*i.e.,* that rarely occur in the corpus) are unlikely to be relevant for the classification task, because they are probably occasional terms or specific slang. We filtered out terms whose frequency is below a certain threshold. We then applied a lower threshold to term frequency. This contributes to the generalizability of our results, since these words are probably not of general use (like a variable name) and could lead to overfitting the training data. When a line contains a word that does not have a correspond-

ing feature in the vector, the word is simply ignored, causing each line to be an extremely sparse vector.

Punctuation

When dealing with documents written in only one language (*e.g.,* NL), IR systems usually remove punctuation; the topic of a document resides in the words, then punctuation is considered as noise that adds no relevant information in a bag of words approach. In this case, where we must distinguish lines written in languages with different syntaxes, we considered punctuation to be an essential aspect to take into account. For example, in Figure 1.1, lines with code (*i.e.,* 17 to 20) have a clearly different frequency of parenthesis if compared to NL parts, and lines with stack traces (*i.e.,* 7 to 11) present more dots. This is because –differently from the natural language– punctuation in categories like code or patch has a syntax that follows a formal grammar (excluding the case where an email can contain some code with error, or where the text has been corrupted by email formatting). This means that the distribution of the punctuation might be representative of a category. Unless the punctuation marks are separated by words or spaces (*e.g.,* the dots in *javax.swing.*, are two occurrences of the feature "."), We considered them as a single term. In this way, we could also recognize the special characters in line 24 (*i.e.,* "@@"), characteristic of a patch block header; or the emoticon in line 29 (*i.e.,* "=)"), a feature of NL or junk. Also in this case, we only considered punctuation marks that appear in at least 100 lines.

With the addition of punctuation marks as features and after the threshold filtering, the considered features grew to 510 elements.

Bi-grams

To use Naïve Bayes means accept the strong assumption that features are conditionally independent. As a result, inside a text, the presence of a word is considered to be unrelated to the presence of its predecessor or successor. In this task, the effect of this strong assumption is mitigated by the fact that we are

considering the emails at row level. This assumption also makes the modeling of NL text features easier: Defining the possible dependencies between words in NL sentences would be an overwhelming task. However, in the other languages considered, which have a stricter syntax, we often find patterns of terms appearing together. For example, "public void" is very frequent in lines of code (*e.g.,* line 17 in Figure 1.1), while "java)" is frequent in stack traces. To model and exploit this intrinsic dependency characteristic of some terms, thus also reducing the negative effects of Naïve Bayes' assumptions, we added *bi-grams* to the set of considered features. A bi-gram is a pair of terms that appear one after the other in a text: For example, in the first line of Figure 1.1, we find two bi-grams: "Alice wrote", "wrote:". We derived all the bi-grams in the corpus and added those more frequent than the threshold as new features. With *bi-grams*, the number of features reached 827 elements.

5.1.2 Training and Evaluation Methodologies

After we defined all the aforementioned features, we modeled each line as a 828-dimensional vector. The first 827 elements of each vector represent the features, while the last one is the manual classification value (*e.g.,* "stack trace"). The features are populated counting the corresponding term's occurrences in the line text, *e.g.,* if the feature t_i corresponds to the term "public", and the line l contains two occurrences of it, then in v_l, we have $t_{i(l)} = 2$. When a line contains terms that are not mapped as feature, these terms are discarded.

Since we decided to use a *supervised* machine-learning algorithm, we needed to train it on classified data. We did 10-fold cross validation, *i.e.,* we split the dataset in 10 folds, using 9 folds (90% of the instances) as training set to build the prediction model, and the remaining fold as a validation set to evaluate the accuracy of the model. Each fold is used once as a validation set. Since the distribution of the classes is not homogeneous (*i.e.,* NL lines are more frequent than stack traces lines), we applied a *stratified* cross validation, in which the distribution of classes is kept equal in both training and test sets. At this point, the starting set for the cross validation was composed of all the lines from all the emails, thus we merged the different mailing lists. The final approach is also evaluated with cross mailing list validation to ensure its generalizability.

To evaluate the effectiveness of the prediction, we measured two IR metrics broadly used in classification [29], namely *precision* and *recall*

$$Precision = \frac{|TP|}{|TP + FP|} \qquad Recall = \frac{|TP|}{|TP + FN|}$$

TP (true positives) are the correctly classified lines, FP (false positives) are the incorrectly classified lines and FN (false negatives) are the incorrectly ignored lines. Since precision (P) and recall (R) trade off against one another, we also report the *F-measure*: their weighted harmonic mean [29]. With *F-measure* it is possible to decide on a different weighting of precision and recall. We chose to emphasize neither recall nor precision by using the *balanced* F-measure.

5.1.3 Term Based Features and Overfitting

By considering the entire set of features (*i.e.,* words, punctuation, bi-grams, and context), we obtained 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 5.1). By reducing the features that are not valuable to correctly predict instances outside the training set, overfitting decreases, while the generalizability of the results increases.

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

Figure 5.2 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 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

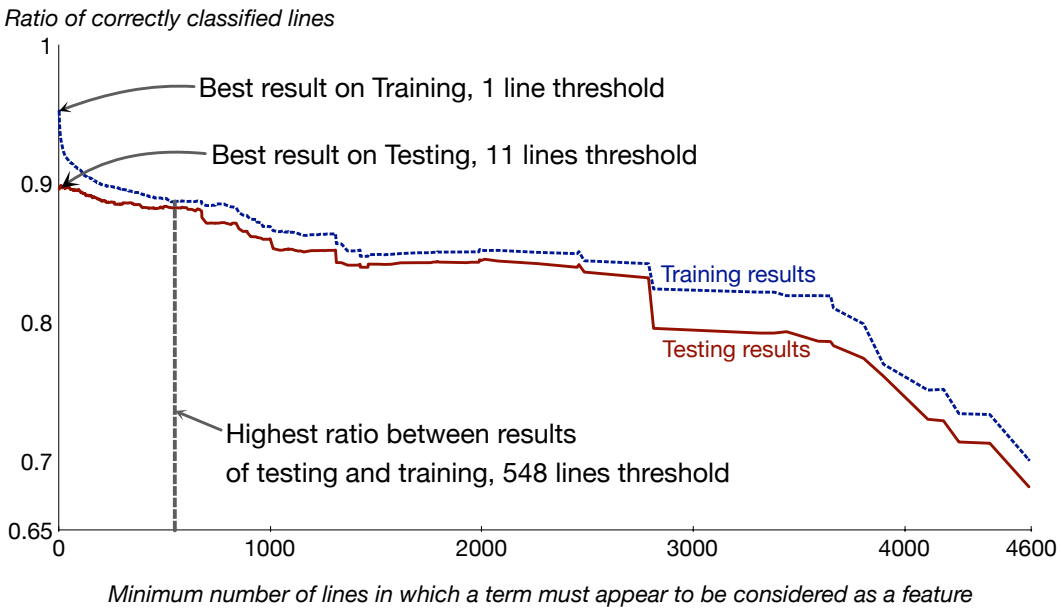

Figure 5.2: Results on training and test sets, by line threshold for features

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 [39]. We found 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.

5.1.4 Results

Table 5.1 shows the results achieved by the Naïve Bayes classifier, which was trained and evaluated with the 10-fold stratified cross-validation. As expected, the chosen terms are already reasonable features for text classification, especially when also considering punctuation and bi-grams as features.

	Average			Correct	
	Precision	Recall	F-Measure	Instances	
Words	0.771	0.735	0.716	49,852	73.5%
Words and Punctuation	0.863	0.860	0.860	58,316	86.0%
Words, Punctuation and Bi-grams	0.876	0.867	0.868	58,806	86.7%

Table 5.1: Results with term based classification, by feature sets

classified as →	NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	31,434	1,006	240	347	114	0.858	0.948	0.901
Junk	4,772	23,943	495	177	198	0.944	0.809	0.871
Patch	219	125	1,130	567	41	0.571	0.543	0.557
Source Code	157	81	112	1,520	44	0.581	0.794	0.671
Stack Trace	69	214	3	4	779	0.662	0.729	0.694

Table 5.2: Classification with words, punctuation, and bi-grams

Table 5.2 reports confusion matrix [29], precision, recall, and F-measure values for the classification done when considering all the term-features (*i.e.,* words, punctuation, and bi-grams). We noticed how patch lines are often classified as code: This is reasonable, considering that we only analyzed features of a single line, thus losing the context. For example, even a human reader could not determine whether line 28 in Figure 1.1 belongs to the patch category or to the code one, without inspecting the preceding lines. From Table 5.2 we also noted that junk, patch, and stack traces are often misclassified among themselves. This is probably due to the punctuation that is similar in these three classes.

5.2 Parsing based classification

In the previous approach, all the features (*i.e.,* terms) considered for classification are extracted *only* from the line under classification, thus taking into account a limited context. Even though the results (see Table 5.2) justify this simplification for most of the cases, some of the considered classes (*i.e.,* patch and stack trace) present a structure that can only be recognized considering a larger number of lines. For example, line 7 in Figure 1.1 cannot be easily distinguished from NL, unless we read the following lines; similarly, line 18 and line 25, which have exactly the same content, can be mapped to the correct class only if considering the surrounding lines.

Researchers dealing with a similar problem tried to solve it by adding new features seizing the characteristics of lines close to the one under classification [14, 37]. To adapt this approach to our case, for each line being classified, we also considered what appears in the preceding and in the following lines. For example, in addition to "@@", we added the features "@@-lineBefore", and "@@-lineAfter". With this information, the classifier was facilitated in recognizing line 25 as part of a patch, especially thanks to the "@@-lineBefore" feature with value 2. By trying this approach, we managed to improve the overall results, but this also led to three main issues:

1. The vectors dimension increased considerably,
2. some patterns cannot be recognized by only considering one line of lookahead,
3. we still use the same kind of supervised approach.

About the first issue, by simply considering the features of only two additional lines, the dimension of the vector tripled and reached 2,481 features. Such a high value hinders the feasibility of testing the features with machine-learning algorithms other than Naïve Bayes, because they mostly have a polynomial complexity and such a high number of feature would severely impact the training time. The second issue could be tackled by considering more lines and features, but this would worse the first issue even more. The third concern regards the

fact that this approach not only requires training data to generate the classifier, but also that it is still based on the same approach: The possible intrinsic flaws of Naïve Bayes on certain instances can hardly be tackled by only considering more features.

For these reasons, we decided to *not* consider features from other lines, but tackle the classification from a different perspective and use a different approach, *i.e.,* parsing. In fact, three of the considered classes (*i.e.,* stack trace, patch, and source code), which are either produced or to be consumed by a machine, present a clearly structured and defined syntax that might be recognized even if embedded in a noisy unstructured context. As illustrated below, the research group had already developed a parsing-based solution to classify email content. We then fine-tuned the parsers for the task, and then we combined them with my machine learning based solution. We devised one specialized parser per each class, excluding NL text. In the following we detail the most salient features of each parser, that we contributed to develop.

5.2.1 Stack trace parsing

Let's define some terminology: The *exceptionMessage* refers to the NL message usually included at the beginning of stack traces (*e.g.,* line 7 in Figure 1.1); the *atLine* refers to a line that reports a method invocation occurred in a specific file (*e.g.,* lines 8 to 11); the *ellipsisLine* is a line used to reduce lengthy stack traces and has the form: "...<number> more"; the *causedByLine* is a line that might appear at any point in a stack trace to introduce a new nested trace and has the form: "Caused by: <stacktrace>".

Among these elements, *atLine*s and *ellipsisLine*s have the most recognizable form. By using the concept of *island parsing* [36] and the parser generator *PetitParser* [33], 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 a mostly unpre-

dictable structure (*e.g.,* different JAVA virtual machine versions may output the same error message differently), thus they cannot be parsed with a specific grammar. To overcome this issue we use a double-pass approach: In the first pass, we recognize and mark all the *atLine*s and *ellipsisLine*s; in the second pass, we look for each line that contains strings such as "exception", "error", "failure", *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*. We empirically found the *n* value equals to 3, to be a good tradeoff between precision and recall.

For example, if we apply our stack trace parser to the email in Figure 1.1, in the first pass, it would classify lines 8 to 11 as *stack trace*; in the second pass, it would consider lines 5 and 7 as *exceptionMessage* candidates, since they both contain the string "exception". Finally, it will only pick line 7, because in the next 3 lines there is an *atLine* element (in this heuristic, we also count the empty lines, such as the line between 6 and 7).

Since this is a method that does not require training, we could test it on the whole manually classified instances. We reached an F-measure value of 99.1% in the classification of stack trace lines. The complete results are reported in the first row of Table 5.3.

5.2.2 Patch parsing

For the patch parser, we also define some terminology: The *patchHeader* refers to the first two lines of a patch, which contain the reference to the modified file and, optionally, the revision versions (*e.g.,* lines 22 and 23 in Figure 1.1); the *patchBlockHeader* refers to the lines detailing the modification done by the patch on a chunk (*e.g.,* line 22); and the *patchBlock* refers to all the lines in the chunk (*e.g.,* lines 25 to 28). We note that a single patch has only one *patchHeader*, while it might have multiple *patchBlockHeader*s followed by the respective *patchBlock*s.

We adopted an approach similar to the stack trace parser: We started from the most recognizable lines and expanded 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*, even if split on multiple lines, by using the tokens "—", "+++", and "@@" as hooks; then we generated a parser that first recognizes the *patchBlockHeader* (thanks to its clear structure), then matches the following *patchBlock*. The patch blocks are problematic, since they have variable length and their ending is not clearly defined. 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. Bird *et al.* tackled the patch block ending issue both by using the information about the range to be found in the *patchBlockHeader* and by analyzing how a line starts (usually the context lines should be preceded by a space) [10]. However, in our dataset we found this information to be not reliable, because of unexpected line breaks and wrong formatting. For this reason, we implemented a lookahead heuristic that checks whether the lines after the "+" or "-" signs might be good candidates as patch. The heuristics checks whether the lines are source code, by using a reduced version of the source code parser described later (Section 5.2.3), and in the positive case it classifies them as *patch*.

With this method we reached an F-measure value of 97.9%. The complete results are reported in the first row of Table 5.3. As expected, since we used a conservative lookahead threshold (maximum four lines), we have a higher precision and lower recall. By manually inspecting the false negatives, we noticed that the low recall is also due to some patch lines that have neither *patchHeader* nor *patchBlockHeader*, thus are completely not considered by our parser.

5.2.3 Source code parsing

Among the three classes with 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, when discussing, usually do not report a complete compilation unit (*e.g.,* an entire JAVA class definition), but only some selected fragments (*e.g.,* the method declaration in lines 17 to 20 in Figure 1.1). These fragments can present more ambiguities, with respect to NL and junk, than other blocks of patches or stack traces. For example, if a line contains only the words "public class", it can be either the beginning of a class declaration or a NL sentence talking about a

specific class. To overcome this issue we wrote a parser based on the concept of island parsing. In our case, however, we did not limit ourselves to define only a few construct of interests, but we wrote a complete JAVA grammar for PetitParser[1], by implementing the *entire* latest specification of the official JAVA language [20]. Then, we wrote an island parser able to recognize most of the constructs of the grammar (we excluded those that are too ambiguous with NL text), starting from the most comprehensive (*i.e.,* compilation unit) down to very specific ones (*e.g.,* expression statements). In addition, we also added rules to recognize incomplete constructs (such as method declarations without the body–common in email discussions).

Compared to BESC [2], a previous approach, this island parser reaches higher precision and is able to locate constructs that span on more lines. For example, BESC would never classify a line with "public class" as code, while our island approach can classify it as code according to the surrounding lines.

We note that our island parser for source code would match most of the content of *patchBlock*s, because they do contain valid source code. This increases the number of false positives. For this reason, we chain the source code parsing to the patch parsing: We first detect the patches, then, on the lines that are *not* classified as patch, we run the source 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.

As expected, in its task of classifying source code, this method reaches a lower F-measure (*i.e.,* 92.6%), compared to the values achieved by the parsers for the previous two classes. This is mainly due to extremely incomplete source fragments that could be only detected by seriously impacting the approach precision.

5.2.4 Junk parsing

Even though noisy text, such as authors' signatures, is difficult to automatically distinguish from NL text without reading the actual content, we found some

[1]Our full JAVA parser is available at `www.squeaksource.com/PetitJava`

peculiar common patterns that can be matched with a parser. This approach is completed in three steps:

1. matching and classification of email headers (such as those in lines 1 and 2 in Figure 1.1) with a simple 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) and authors; and
3. usage of the recognized signatures to automatically compose a grammar for generating a parser to match them, in any possible formatting or position in the email body.

To recognize signatures in step 2, we consider all the emails whose last block of text is not quoted from previous emails. In these emails, the authors themselves conclude the message and most probably include their signatures. For example, the email in Figure 1.1 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 lines 34 up to line 16). Every time we encounter a line that starts with, or is only composed of, two dashes or more than 3 (to avoid matching patch headers) dashes, underscores, or stars, we take out the lines up to the bottom and consider this block to be a signature. We continue the process until we reach the top of the not quoted block. For example the algorithm, applied to the email in Figure 1.1, would extract lines 32 to 34, and line 31 as block signatures.

However, by simply classifying these blocks as junk, we would miss the cases in which signatures compare in quoted text (such as in lines 14 and 15). For this reason, we conduct the aforementioned step 3: We use each extracted string to automatically define a grammar able to recognize the signature in any possible position or formatting the text (*e.g.,* a signature can be quoted with wrong line breaks). We use these grammars to automatically generate the corresponding parsers with PetitParser. Finally, we classify as *junk* all the matched lines.

This approach reaches an F-measure value of 81.2%, by recognizing more than 65% of the junk lines.

5.2.5 Results

	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

Table 5.3: Single classification results achieved by using parsers

Table 5.3 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 described in Section 5.2.1 to classify lines as *stack trace*. The false positives (*e.g.,* four in the first row), are instances that are classified as *stack trace* by the method, but had a different manual classification, such as *junk*.

We argue that these parsers are not only valuable thanks to the high classification values they achieve, but also because they are mailing list independent and require no training to be used. The former feature is given by the fact that the parsers rely on syntactical characteristics of programming languages, stack traces, and patches, that are the same across all the mailing list pertaining to JAVA systems; the latter feature allows their usage on any textual source of data.

However, these parser-based approaches have limitations. First, 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. In addition, the values reached in classifying junk are lower than those achieved with the term based approach, which was able to find junk with an F-measure of 87.1%. In the next we present the method we devised to merge machine-learning and parser-based approaches, with the aim of overcoming these issues and create a more complete, precise, and robust approach.

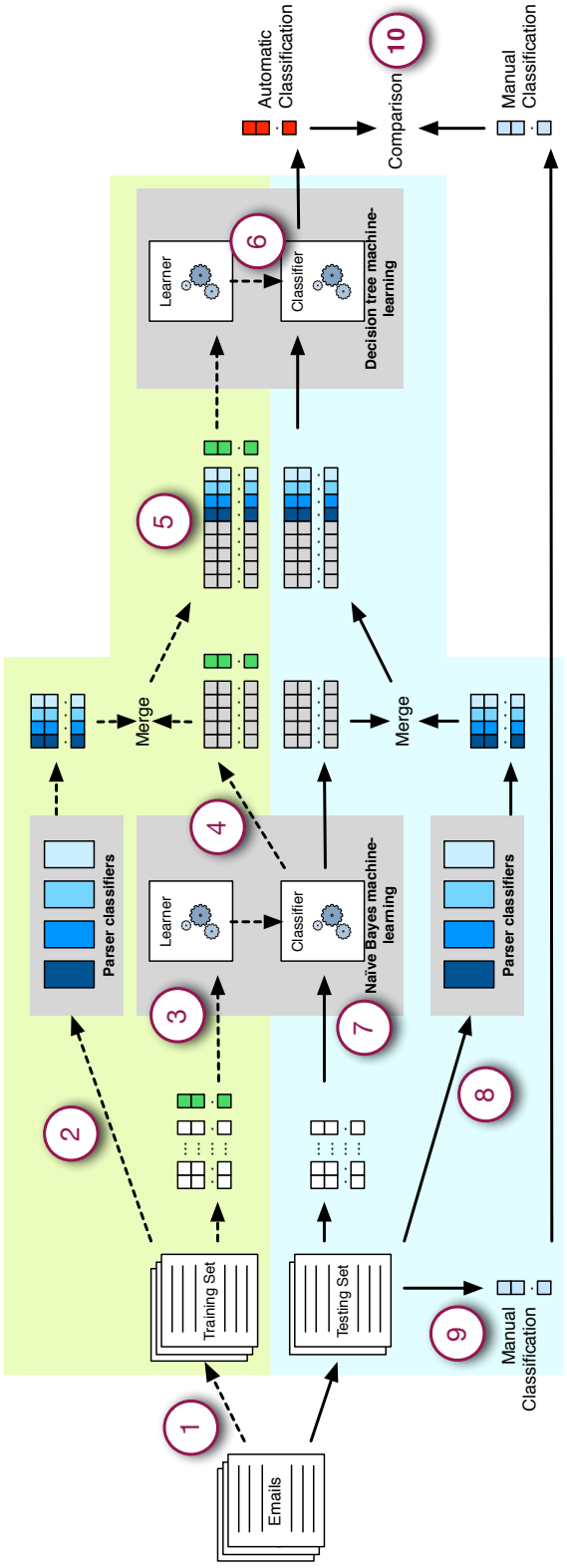

Figure 5.3: Training and Test Process of the Unified Classification Approach

5.3 Mixed Approach

As explained before, the main goal was to integrate the parsing based classifiers with machine learning techniques, to produce a better and more robust classification process. So we merged the two approaches into a single method, generating a *mixed* approach. This approach merges the valuable characteristics of the term based classification and the parser-based approaches.

5.3.1 Adding parsing results to Naïve Bayes

Similarly to most of the other machine-learning approaches, Naïve Bayes is not limited to consider *terms* as features. We then considered other aspects that could be relevant for the classification to include them as new features in the classification process. For example, researchers considered the quotation level in which an email line reside as a valuable feature for recognizing noise [37].

We had to decide how to merge the parsers' results with the features I had extracted. Given these premises, we decided to add the output of the parser-based classification to improve the Naïve Bayes machine-learning process. To achieve this, we joined the four prediction of the parsers together with the features of the words, punctuation and bi-grams, to form a new vector. Each new feature maps the output of a parser: Its value is one when the corresponding parser-based classifier matches the specific line, and it is zero in the other cases. For example, we expect the feature-vectors corresponding to lines 31 to 34 in Figure 1.1 to have the feature "junk parser" with value one, while the other parser-based features with value zero; similarly, we expect the "stack trace parser" feature to be one for the vector of line 10, while the others to be zero on the same line. It would be possible that, in a few cases, more than one parser-based features have value one.

With these new features in place, we used again the Naïve Bayes machine-learning process and conducted training and evaluation as presented in Section 5.1.2.

5.3.2 Results

Table 5.4 reports the confusion matrix on the results achieved by adding the four parser-based features to the Naïve Bayes approach. Overall, it correctly classified 61,875 instances (91.3%), 3,000 instances more than the previous machine-learning approach.

classified as →	NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	31,603	1,001	184	244	184	0.902	0.954	0.927
Junk	3,320	25,468	416	176	205	0.960	0.861	0.908
Patch	51	23	1,969	32	7	0.759	0.946	0.842
Source Code	57	22	23	1,787	25	0.797	0.934	0.860
Stack Trace	9	9	1	2	1,048	0.752	0.980	0.851

Table 5.4: Results adding parser-based features

Comparing the confusion matrices of the machine-learning approaches (Table 5.4 and 5.2), we saw that the new features helped to decrease the instances wrongly classified as NL text. Being NL the most frequent class (see Table 4.2), it has a strong impact on the evaluation of the MAP hypothesis of Naïve Bayes; 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 can note that the class *patch* has more than a hundred wrongly classified instances. This contradicts the precision value of one reported by the single patch parser classifier. This is probably due to the fact that, even if the "patch parser" feature has a very high weight in the computation, it is placed at the same level of the other features that, being a large number, also influence the results. Although this is good for the junk class, we do not find it optimal in all the cases. Another issue of this approach is that the number of considered features is very high and makes it hard to try other machine-learning approaches. In fact, it is clear that the conditional independence assumption, that seemed reasonable when considering the words in a line, is now inadequate dealing with the parsers' classifications. Also, Naïve Bayes cannot sufficiently enhance

the weight of the parsers, with respect to the word features. It then would be possible that an approach not having the conditional independence assumption of Naïve Bayes could be better suited for modeling the new features, which are highly dependent.

Motivated by these issues, we devised a two-passes classifier approach to better use parser-based classifiers, but still exploiting Naïve Bayes qualities.

5.3.3 Unified Classification Approach

The idea behind this approach was to use Naïve Bayes to evaluate a partial classification only on the features based on *terms*, and then use another machine-learning classifier to merge and model Naïve Bayes results and parser-based classifications.

Training

We first (Point 1) extracted the emails on which we wanted to train the machine-learning algorithms (*e.g.,* 90% of the sample set for the stratified 10-fold cross validation). Then, we concurrently provided them in input to the parser-based classifiers (Point 2), and for each line we extracted the feature-vector on words, punctuation, and bi-grams, that we provided to the Naïve Bayes learning algorithm, along with the manual classification (Point 3). With this data, Naïve Bayes trains a classifier, very similarly to what we did in my first term based approach. But instead of returning the instance classifications, it outputted a 5-dimensions vector for every line: Each dimension represents a class (*e.g., NL text*) and the value is the probability evaluated by Naïve Bayes on that instance to belong to that class. In other words, instead of simply picking the highest value, and provide the final classification, we output all the 5 probabilities and we map them to features, thus reducing the initial 827 features to 5 predictions.

At the same time, the parser-based classifiers created other four features, as in Section 5.3.1. Once both feature sets were evaluated, our approach merged

them and constructed a unified 9-dimensions vector, to which it added the manual classification (Point 5). This final vector was the input to another machine-learning algorithm to train the final classifier (Point 6): The actual output of the training.

The choice of the machine-learning technique for the second step is critical: We need an algorithm that correctly models the peculiar characteristics of our features. Thanks to the reduced dimensions of the new feature-vectors, we was able to try a number of different machine-learning approaches. The two methods that we empirically saw obtain better results were *Support Vector Machines* and *Decision Trees*, which both are broadly used methods in data mining (see Section 3.2). The two methods perform nearly identical in time and results, with the latter performing slightly better than the former. We found the simple decision trees to be the most suited to our parser-based classifiers because they output almost mutually exclusive features that permit a reading and interpretation of the features that had the larger influence on the classification. SVM, however, would probably benefit from a selection of a kernel specifically chosen for the task, which would represent an interesting aspect for further refinement of the method.

Testing

The test process is depicted in the bottom half of Figure 5.3. We took the testing set of emails (*i.e.,* 10% of the sample set in the stratified 10-fold cross validation) and we extracted the manual classification. Then, we concurrently provide the emails to the parser-based classifiers (Point 8) and created the feature-vectors, to be given as an input to the previously trained Naïve Bayes classifier (Point 7). Subsequently, we merged the output of the two technique in a unified 9-dimensions vector which I input to the second machine-learning classifier, previously trained, which outpus the final classification. We compared this classification (Point 10) to the manual one (Point 9) and we evaluated the results.

As final evaluation, the training and test phases were repeated 10 times according to the cross validation.

5.3.4 Results

classified as →	NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	30,719	2,363	1	57	2	0.959	0.927	0.943
Junk	1,210	28,252	35	84	4	0.921	0.955	0.938
Patch	16	35	2,019	12	0	0.981	0.970	0.975
Source Code	88	28	3	1,795	0	0.921	0.938	0.929
Stack Trace	15	10	0	0	1,044	0.994	0.977	0.985

Table 5.5: Unified approach results on 10-fold cross validation

Table 5.5 presents the results achieved by our unified classification approach. This two-steps approach, which differently merges and model the available information, improves the results for all the classes, by increasing not only the results related to the parser-based classifiers (*i.e.,* patch, stack trace, and code) but also those connected to the Naïve Bayes algorithm. The F-measure values are all increased, even though there is 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.

Chapter 6

Discussion And Threats to Validity

6.1 Discussion

The results of the unified classification approach were particularly good and exceeded our expectations. However, the effectiveness of the method is strictly related to the terms that it indexes, and these terms are depending on the kind of mailing list that we consider, as every group of user may represent a completely different environment. In fact, we have seen that two mailing lists, even if treating similar projects for programming language, may differ substantially in their content. For example, the mailing list of ArgoUML have a lot of emails related to the architecture of their software, or that discuss about a bug in a Java class, while the mailing list of Freenet is much more concerned about privacy, its political implication, freedom or implementation of cryptography algorithms.

Obviously different topics mean different distribution of words and different term frequencies, so that the *model* of a category built using a set of mailing lists may not be representative of the distribution of a new mailing list. It is then legitimate to expect that the results previously obtained should be negatively effected and may perform poorly when applied to a brand new mailing list.

To test the generalizability of the results achieved, we then performed a

"mailing list cross validation". we created 4 different folds, where each fold consisted of all the emails from a mailing list. We then used three of these folds as the training set to train the classifier and the remaining one to test the classification predicted by the method, so that the emails used to test the model were completely separated from the emails used to train it.

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 conducted this four times rotating the mailing lists, and then measured the average results. As explained above, the considered mailing lists only share the programming language, *i.e.,* JAVA, thus we expect this to be a veritable test to understand which results we could expect from the classifier applied to a new and unseen JAVA development mailing lists.

classified as →	NL Text	Junk	Patch	Source Code	Stack Trace	Precision	Recall	F-Measure
NL Text	30,499	2,578	1	62	2	0.945	0.920	0.933
Junk	1,591	27,876	36	78	4	0.913	0.942	0.927
Patch	63	48	1,945	26	0	0.980	0.934	0.956
Source Code	106	37	3	1,768	0	0.914	0.924	0.919
Stack Trace	6	8	0	0	1,055	0.994	0.987	0.991

Table 6.1: Results on mailing list cross validation

Table 6.1 reports the results on this custom cross validation achieved using the unified classification approach. As expected, the values are lower to those achieved with stratified 10-fold cross validation. This suggests that this test might provide more reliable information when compared to a real world usage scenario on a new mailing list. However, although the decrease of precision and recall, they still remain above 91%, with an average F-measure of 94.5%. This shows an important result: Even when changing a system, which also means changing the terms used in the emails and their distribution, the generated model is robust enough to perform reasonably well without the need of additional training on new data.

We think that in this result, a big role is played by the use of threshold, bigrams and parsers. In fact, the effect of the different distribution of the terms

exposed above is not enough to overcome the clustering effect given by the different syntax of each category. As such, it can be considered as noise that is removed during the cleaning phase. The remaining terms are then a sort of *cliques*, representative of each category.

6.2 Threats to Validity

6.2.1 Construct Validity

Threats to construct validity 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 MAILPEEK, a web application to ease the annotation process (see Chapter 4). The classification process was cross-inspected, checking 10% of the emails finding only 12 erroneously classified lines. We corrected these 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.

6.2.2 Statistical Conclusion

Threats to statistical conclusions are concerned with whether the dataset is large enough to support the claims. We took samples of email populations that were representative with a 95% confidence and a 5% error level, which are standard values. On the number of lines, the corpus has 67,792 not empty classified lines.

6.2.3 External Validity

Threats to external validity are concerned with the generalizability of the results. The approaches we tried may show different results when applied to other soft-

ware systems and mailing lists. We showed that the mailing list cross validation performs well with new mailing lists, developed by separate communities still reports significative results. A problem may arise when considering a different programming language. In this case a new classification and training phase and new parsers should be necessary. The mixed programming language case still remains to verify.

It is also important to note that we only considered open-source systems, and that usage patterns may vary in the industry. In particular, mailing lists often occupy a central role in the development of open-source systems, which may not be the case in systems developed in a more centralized environment.

Chapter 7

Conclusion

The idea behind this project is that development mailing lists contain significant information about the software system they discuss. However, this information is buried inside the archives. In order to render this information available, we dealt with the problem of accurately classifying email data content according to five main language categories:

1. Natural language text
2. source code fragments
3. stack traces
4. code patches
5. junk

To devise a robust approach to conduct this classification and prove it correctness, we first carefully conducted the manual classification of statistically significant sample sets of emails extracted from four development mailing list: ArgoUML, Freenet, JMeter, and Mina.

We presented Mailpeek, a web application developed to assist the data set construction in its two main steps:

1. automatically fetching the *Markmail* online service and download a formatted version of publicly available mailing lists;

2. supporting the manual classification task providing a user friendly interface that easily drove the user through the tagging of the email content.

On the basis of the obtained data, We established that a classification at line granularity offered a reasonable accuracy. We conducted an experiment where we tested different machine learning techniques, and evaluated them in terms of precision, recall, and F-measure. We investigated how to merge my classifier with the parsers based on island-parsing, developed by the research group. Since these techniques in isolation are neither accurate nor robust enough to deal with the noisy email data, we combined them in an unified 2-steps approach that tries to overcome the limitation of the techniques if taken alone. The results obtained are very positive, even when performing cross mailing list evaluation. The reason is that the parser-based classifiers are mailing list independent and offer a valuable basis. On the other hand, the probabilistic machine-learning approach, gains great benefits from the data cleaning phase, thus making the method even more robust and general.

All data sets of the experiment and a SMALLTALK image containing a working copy of the classifier with full source code are available at the website located at `http://mucca.inf.usi.ch`.

7.1 Contribution

The contributions of our work are:

A new dataset containing 1,493 development emails.
We built a dataset containing classified email from four development mailing lists. This data can be used for future analysis on information extraction from software repositories.

A web application to download and classify mailing lists.

We developed MAILPEEK, a web application written in SMALLTALK to automatically download mailing lists from the *MarkMail* service. The application also supports the manual email classification with an user friendly graphical interface.

A new approach for fine-grained email classification.

We devised a new approach that combines parsing methods with machine learning techniques, to precisely classify the content of development email at row-level. Researchers have shown that a precise pre-processing data cleaning phase could greatly improve the performance in an analysis method [9]. Since all the previous works considered the problem of email classification only ad document scope, our method is the first that presents such a fine-grained resolution, and that allows accurate analysis on email content.

The evaluation of our method tested on a real dataset.

We tested our approach on our dataset built from real development mailing lists. We obtained surprisingly good performances, with a f-measure above 0.9 for each category.

7.2 Exploitation

This work allows accurate analysis and data cleaning on development emails. In particular, the method will be used to obtain clean emails and perform specific analysis on each category, with the intent of extraction informations that describe the software system to which they are related (*e.g.,* the evolution of a system, or make predictions about bugs in the code).

This thesis work also produced a conference paper, publshed at ICSE 2012 a top tier conference [5].

7.3 Future work

The first improvement involves the extension of the classification categories, adding new group. For example, the category *link* is not considered in this work, but still contains valuable information, since it defines a form of correlation between two topics that could indicate an interaction between two classes or additional information about a bug.

The second idea is on reconsidering the use of Support Vector Machines. During the experiment presented in this work, SVM were adopted with standard kernels, leading to similar results to the Decision Trees. It would be interesting to explore the behavior of this approach by fine-tuning a *ad hoc* kernels that could better fit the context.

The third possible expansion involves the classification of the emails. I, together with a PhD student, have spent a lot of time reading and classifying a lot of emails. An approach considering techniques of semi-supervised learning could be applied to allow the generation of new examples, thus permitting more precision and shorter starting time while classifying new email, for example when applying the approach to a new domain like a new programming language.

7.4 My contribution to the experiment

When I joined, the group was trying to perform email data cleaning using a pure parser approach. My involvement into the project consisted in integrating these parsers with machine learning techniques, to obtain more robust classifiers. In particular, my contribution consisted in:

Analysis of the features to include in the method. The first step required an analysis of the domain, to identify the exploitable characteristics of the data we wanted to analyze.

Development of MILER2 I wrote a large part of the code of MILER2, the application to download development emails from *MarkMail*

Development of MAILPEEK I write *MailPeek*, the web application to manually classify the email content.

Manual classification of the dataset I, together with a PhD student, manually classified the emails that we used in our dataset.

Selection of the machine learning methods Based on the analysis of the domain, I selected some machine learning methods that I proposed to the group to be tested on the dataset.

Development of the machine learning methods using WEKA I performed the tests with WEKA to find the methods that best fitted our problem.

Help tuning the parsers to be used in the mixed approach I had a little part in the tuning of the parsers to obtain the best performances used together with machine learning algorithms.

Development of the unified approach I proposed some approaches to develop the unified method, and I tested them.

Help writing the paper I was involved in the writing of the paper, and had the possibility of contribute in this exciting experience.

Each step of this process was assisted by the members of the group, which helped me in the evaluation of the approaches.

7.5 Epilogue

The experience of the thesis helped me to improve many aspect of my computer science education. First of all, being able to work in a group motivated by passion and enthusiasm, inspired me to develop the experiment with true motivation and curiosity, and tested my relational abilities dealing with other people to constructively participate in finding a solution in a collaborative environment. Furthermore, the need to tackle a real problem from the beginning to the implementation details, made me realize some aspects of the analysis process that I did not get so crucially during the MSc courses. Particularly, I focused

on modeling the problem in a significant way, choosing the correct features and synthesizing the data in an efficient and sustainable form, whereas during the classes I perceived a bigger attention towards the algorithms mechanic itself.

I also learned SMALLTALK, which has been a great experience either from an historical point of view—since it is the language that introduced many concept considered essential in modern programming languages—and also as a different and interesting style of thinking in Object Oriented Programming, which widened my comprehension and interest for design patterns.

The REVEAL research group is mainly interested in research about reverse engineering, mining software repositories, software evolution and software visualization. This experience allowed me to explore a field of knowledge that I have since then ignored, and made me develop an interest for the application of the artificial intelligence techniques that I have studied during my education to these topics. Moreover, being part of the process of writing a paper has been an exciting experience, that gave me an insight on how research is conducted; an experience that I recall with great satisfaction.

I think that the instruments I got from my education were adequate for the task, and allowed me make a concrete and significant contribution.

Bibliography

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] Alberto Bacchelli, Marco D'Ambros, and Michele 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.

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

[4] Alberto Bacchelli, Michele Lanza, and Marco D'Ambros. Miler: A toolset for exploring email data. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, page to be published, 2011.

[5] Alberto Bacchelli, Tommaso dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385. IEEE CS Press, 2012.

[6] Matthew J. Beal, Zoubin Ghahramani, and Carl Edward Rasmussen. Factorial hidden markov models. In *Machine Learning*, pages 29–245. MIT Press, 1997.

[7] Adam L. Berger, Vincent J. Della Pietra, and Stephen A. Della Pietra. A

maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[8] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *Proceedings of MSR 2008 (5th IEEE Working Conference on Mining Software Repositories)*, pages 27–30. ACM, 2008.

[9] Nicolas Bettenburg, Emad Shihab, and Ahmed 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 Computer Society, 2009.

[10] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in OSS projects. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, pages 26–29. IEEE Computer Society, 2007.

[11] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? Bias in bug-fix datasets. In *Proceedings of FSE 2009 (17th ACM International Symposium on Foundations of Software Engineering)*, pages 121–130. ACM, 2009.

[12] B. E. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

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

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

[15] Marco D'Ambros, Michele Lanza, and Martin Pinzger. The metabase: Generating object persistency using meta descriptions. In *Proceedings of FAMOOSr (1st Workshop on FAMIX and MOOSE in Reengineering)*, 2007.

[16] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31–40. IEEE CS Press, 2010.

[17] Alex Dekhtyar, Jane Huffman Hayes, and Tim Menzies. Text is software too. In *Proceedings of MSR 2004 (1st International Workshop on Mining Software Repositories)*, pages 22–26, 2004.

[18] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5): 56–63, 2007. ISSN 0740-7459.

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

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3nd edition, 2005.

[21] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 223–226. ACM, 2010.

[22] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[23] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. 2003. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[24] Karen Spärck Jones. Automatic summarising: The state of the art. *Information Processing and Management*, 43:1449–1481, 2007.

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

[26] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineeering)*, page to be published, 2011.

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

[28] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design*. Rockport, 2003. ISBN 978-1-59253-007-6.

[29] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[30] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[31] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

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

[33] Lukas Renggli, Stéphane Ducasse, Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *Proc. of DYLA 2010 (4th Workshop on Dynamic Languages and Applications)*, 2010.

[34] Peter C. Rigby and Ahmed E. Hassan. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, pages 23–. IEEE Computer Society, 2007.

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

[36] Oliviero Stock, Rino Falcone, and Patrizia Insinnamo. Island parsing and bidirectional charts. In *Proc. of the 12th Conf. on Computational Linguistics*, pages 636–641, 1988.

[37] Jie Tang, Hang Li, Yunbo Cao, and Zhaohui 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.

[38] Ahmed E. Hassan Thanh H. D. Nguyen, Bram Adams. A case study of bias in bug-fix datasets. In *Proceedings of WCRE 2010 (17th IEEE Working Conference on Reverse Engineering)*, pages 259 –268. IEEE CS Press, 2010.

[39] Mario Triola. *Elementary Statistics*. Addison-Wesley, 2006. ISBN 0-321-33183-4.

[40] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.