Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests

Giuliano Antoniol and Kamel Ayari[1] Massimiliano Di Penta[2]
Foutse Khomh and Yann-Gaël Guéhéneuc[3]

[1] SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada
[2] RCOST, University of Sannio, I-82100 Benevento, Italy
[3] Ptidej Team – GEODES, DIRO, Université de Montréal, Québec, Canada

E-mails: {giuliano.antoniol,kamel.ayari}@polymtl.ca,
dipenta@unisannio.it, {foutsekh,guehene}@iro.umontreal.ca

Abstract

Bug tracking systems are valuable assets for managing maintenance activities. They are widely used in open-source projects as well as in the software industry. They collect many different kinds of issues: requests for defect fixing, enhancements, refactoring/restructuring activities and organizational issues. These different kinds of issues are simply labeled as "bug" for lack of a better classification support or of knowledge about the possible kinds.

This paper investigates whether the text of the issues posted in bug tracking systems is enough to classify them into corrective maintenance and other kinds of activities.

We show that alternating decision trees, naive Bayes classifiers, and logistic regression can be used to accurately distinguish bugs from other kinds of issues. Results from empirical studies performed on issues for Mozilla, Eclipse, and JBoss indicate that issues can be classified with between 77% and 82% of correct decisions.

This work has been partly funded by NSERC Canada Research Chair Tier I in Software Change and Evolution and a NSERC Discovery grant.

1 Introduction

Open-source as well as closed-source projects manage issues[1] using Bug Tracking Systems (BTS). BTS should be used mostly to manage issues related to corrective maintenance, *i.e.*, bugs. Yet, they often contain entries concerning other software activities, such as perfective or preventive maintenance; request for new features; legal and licensing issues; discussions about architectural changes and restructuring.

The mixing of different kinds of issues in BTS can be easily observed by skimming through the titles of the issues submitted. For example, in Mozilla BTS under the category "Hot Bugs". As of the 14^{th} of May 2008, "Hot Bugs"[2] include bugs such as "Arrow keys stop working after going back one page" (issue number 430723) and other kinds of issues such as "Bring back the advanced search options from v2 to Remora" (request for enhancement, issue number 372841) and "disc" should be "disk" (translation problem, issue number 273267).

Therefore, BTS are used as centralized blackboards where developers and users discuss and record decisions about various issues and doc-

[1] Such issues are commonly called "bug reports" or "bugs" but we avoid using these terms because issues are *not* always bugs, as we will see in the following.

[2] Mozilla "Hot bugs" are available at `https://bugzilla.mozilla.org/duplicates.cgi?sortby=delta&reverse=1&maxrows=100&changedsince=30`.

ument corrective maintenance as well as other activities. In the following, we refer to a corrective maintenance request as a *bug* while other BTS issues, such as perfective and adaptive maintenance, refactoring, discussions, requests for help, and so on, are referred to as *non bug*.

This paper tackles the problem of classifying issues into two classes: bugs and non bugs, with the goal of building automatic classification systems. Such automatic classification systems, or *classifiers*, can be used for assigning bugs to developers, building error proneness models, and effectively scheduling other activities such as enhancement or restructuring.

We choose to use alternating decision trees (an extension of classification trees), naive Bayes classifiers, and logistic regression to build classifiers. Our choice is motivated by the observation that these machine learning techniques produce classifiers more easily interpretable at the price of training the classifiers on a set of pre-labeled data. Five software engineers manually classified BTS entries randomly extracted from resolved issues of Mozilla, Eclipse, and JBoss. On the manually tagged corpus, we built classifiers and evaluated their accuracy (precision and recall) via cross validation.

The contribution of this paper is three-fold:

1. We report our experience in classifying manually 1,800 issues extracted from the BTS of Mozilla, Eclipse, and JBoss using simple majority voting.

2. Using the previous manual classification, we answer three research questions:

   - *RQ1: Issue classification.* To what extent the information contained in issues posted on bug tracking systems can be used to classify such issues, distinguishing bugs (*i.e.*, corrective maintenance) from other activities (*e.g.*, enhancement, refactoring...)?
   - *RQ2: Discriminating terms.* What are the terms/fields that machine learning techniques use to discern bugs from other issues?
   - *RQ3: Comparison with grep.* Do machine learning techniques perform better than *grep* and regular expression matching in general, techniques often used to analyze Concurrent Versions Systems(CVS)/SubVersioN (SVN) logs and classify commits between bugs and other activities?

3. While answering the previous research questions, we build classifiers for BTS issues with between 77% and 82% of correct decisions. We also provide evidence that a large number of issues stored in BTS is not related to corrective maintenance and thus to bugs, putting in perspective previous work on BTS and defect estimation.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 summarizes background notions on the BTS and machine learning techniques. Section 4 describes our research questions, the objects of our study, and the process applied to building the Oracle. Section 5 reports the results of the study and the answers to our questions. Finally, Section 6 concludes the paper and introduces future work.

2 Related Work

Project and source code metrics, architectural and design features are used as independent variables in models explaining or predicting defects. The prediction can be performed at different levels of granularity, i.e., can be related to units such as components, classes, files, functions, or methods; the number of defects contained in the unit or the probability that the unit contains at least one defect is almost always the dependent variable [15, 17, 21, 23, 25]. The underpinning assumption is that an accurate defect prediction can profitably direct verification and validation activities to the subset of artifacts with a higher error proneness.

As previously shown by Basili *et al.* [5] and Gyimothy *et al.* [15], *multivariate logistic regression* [1] represents an effective technique to study the relationship between the fault proneness of classes and source code metrics.

In recent years, the literature reported contributions on merging data from CVS and bug reports to identify whether CVS changes are

related to bug fixes, to detect co-changes and to study evolution patterns.

Fischer *et al.* [11] discussed their experience in populating a relational database with data from Mozilla CVS and bug report repositories. To trace CVS changes to bug reports, they used a pattern matching approach to identify CVS messages related to bug fixing. They investigated the impact of qualified release history data on different source code model entities [10], also using runtime data to build the source code model. They concluded on the possibility of reconstructing information about the evolution of systems but underscored the need for the CVS to support some formal mechanism for linking detailed modification reports and classification of changes.

Sliwerski *et al.* [21] introduced a refined approach to identify whether a change induced a bug fix. They combined a syntactic analysis, *i.e.*, pattern matching, with semantic analysis. Semantic analysis compared the author's name of the CVS change with that of the developer responsible to propose bug fixing in Bugzilla. Consistency of dates and file versions were also part of their heuristics. They found that the larger a change, the more likely it is to induce a fix. They also found that in the Eclipse project, fixes are three times as likely to induce a later change than ordinary enhancements.

The idea of logical coupling between source code artifacts based on data extracted from release history was introduced by Gall *et al.* [13]. Ying [28] presented an approach for the identification of logical coupling that computes association rules among files by applying data mining techniques on CVS repositories. At the same time, Zimmermann *et al.* [29] proposed a change-coupling identification approach capable of recovering fine-grain co-changing entities (*e.g.*, classes, methods, fields). They predicted future changes by detecting causal couplings between entities and assessed change impact to prevent incomplete changes. German [14] abstracted co-changing files into modification requests and analyzed their interrelationships and authors.

Weißgerber *et al.* [25] discussed whether some types of co–changes, in particular refactorings, are less error prone than other types of changes, identifying cases in which this was true and cases in which it was not.

Wang *et al.* [24] proposed an approach to assist triagers in detecting duplicate issues. Their approach combines both natural language information and execution information in the detection. They employed two heuristics to combine the two kinds of information. They calibrated and evaluated their approach on bug reports from the Eclipse and Firefox repositories and concluded that it performs better than the best performance of approaches using only natural language information. However, their approach depends on execution information that are often absent and, if present, costly to obtain and manage because it must be treated manually.

The work presented in this paper builds upon previous contributions, mainly on the work of Sliwerski *et al.* [21] to study the consistency of data from different sources. No previous work deeply investigated the kinds of data stored in BTS, such as the Mozilla's Bugzilla BTS. The main contribution is therefore in the answers to the research questions formulated in the introduction. Although, as any experimental study, we must be wary of some possible threats to the validity of our results (cf. Section 5.4), reported results put into perspective approaches that combine data from CVS and Bugzilla repositories to build quality models: only a subset of issues (about a half) posted on Bugzilla are real bugs, *i.e.*, related to corrective maintenance.

3 Background

This section provides background information about the two BTS from which we downloaded issues to be classified, and about the machine learning techniques used in this paper.

3.1 Bug Tracking Systems

A standard chain of "reaction" to bugs is the following: the client, analyst, or developer makes an *error* while describing the problem or developing software artifacts. This error may or may not result in a *fault* in the system, an unexpected state. This unexpected state may

or may not result in a *bug*, a visible and non-desired event from a user's point of view. When a bug is discovered, it manifests itself, it is documented and details are posted on BTS.

There exist several BTS, more generally known as ticket tracking systems, but Bugzilla[3] and Jira[4] are among the two most popular due to their use in major open-source projects, such as Mozilla, Eclipse, and JBoss. Both Bugzilla and Jira are Web-based systems, offering two principal user interfaces: an interface to consult the list of stored issues and an interface to post and reply to issues. BTS are just front-end to databases and can be queried in different ways. Bugzilla program model is based on CGI-BIN and can be easily queried via HTML `get`; Jira supports RSS and, thus, any RSS reader with an XML parser can be used to extract data.

When posting an issue, most BTS offer a set of fields to label the issue, including severity; keywords; the product, component, version, hardware, and operating system against which the issue is filled. Neither Bugzilla nor Jira make a real distinction between error, faults, bugs and their synonyms, such as defects, crashes, problems. For example, all Bugzilla issues are referred to as *bugs* and only the "Severity" field allows the tag "Evolution".

Bug life cycle somehow depends on the specific bug tracking tool; however, similarities can be identified. When a new bug is discovered, its life begins in the "New" or "Unconfirmed" status. At this stage, the bug is assigned a unique identification number (ID) as well as some properties such as severity, priority, components it affects, discovered and reported time-stamps [3].

Subsequently, a programmer takes the lead or is assigned to the task of proposing a fix. Bug fixes are often submitted as Unix patches or context diff. Patches are associated to bugs via an attachment table linked to the bug ID. Once a bug resolution is proposed and approved, its status reaches the final disposition of "Close". However, at least for the projects considered in this paper, a bug almost never reaches the "Close" state: once it is tagged as "Resolved", it remains forever in this state.

[3] `http://www.bugzilla.org/`

[4] `http://www.atlassian.com/software/jira/`

3.2 Machine Learning Techniques

A *classifier* is a function $f\colon \mathbb{R}^d \mapsto C$ that assign a label from a finite set of classes $C = \{c_1, \ldots, c_q\}$ to observations $\mathbf{x} \in \mathbb{R}^d$. In this paper we are interested in the family of binary classifiers where there are only two classes and thus C contains only two symbols $C = \{0, 1\}$ or $C = \{non\ bug, bug\}$.

Three families of machine learning techniques are available to build a classifier: unsupervised learning, supervised learning, and reinforcement learning [18, 2]. Unsupervised learning, for example clustering algorithms, classifies available data based on some fitness or cost function: often a distance or similarity. Supervised learning, *e.g.*, ClAssification and Regression Trees (CART), assumes that a training set of labeled data is available. A classifier is then built by maximizing some gain or minimizing a cost function, representative of the accuracy of the classifier with respect to the a-priori classification. In reinforcement learning, a user is required to decide if the classification for the current piece of data is correct; the classifier then incrementally learns a classification function. This later family of techniques is not well suited for off-line classification but has been successfully applied in traceability recovery [16].

Unsupervised learning techniques are appealing because no pre-labeled data is needed; however, it is very difficult to interpret the resulting classification and it may be hard to derive guidelines linking the classification with characteristics of the data or of the development process.

Supervised learning techniques, in particular algorithms such as CART, Bayesian classifiers, or logistic regression, produce classifiers more easily interpretable but require a labeled corpus. A labeled corpus is a set of pairs (observation, label) assumed as are random variables $(\mathbf{X}, Y)$ drawn from a fixed but unknown probability distribution μ. The objective of the learning techniques is to find a classifier f with a low error probability $\mathbb{P}_\mu[f(\mathbf{X}) \neq Y]$.

Both the selection and the evaluation of f must be based on some data set D_n containing n labeled pieces of data because the data distri-

bution μ is unknown. Therefore, D_n is usually split into two parts, the *training sample* D_m and the *test sample* D_{n-m}.

A *learning algorithm* is a method that takes the training sample D_m as input and outputs a classifier $f(\mathbf{x}; D_m) = f_m(\mathbf{x})$. A common learning method chooses a function f_m from a function class that minimizes the *training error*

$$L(f, D_m) = \frac{1}{m} \sum_{i=1}^{m} I_{\{f(\mathbf{x}_i) \neq y_i\}} \qquad (1)$$

where I_A is the indicator function of event A.

Examples of learning algorithms using this method include the back propagation algorithm for feed-forward neural nets [7] or the C4.5 algorithm for decision trees [19].

To evaluate the chosen function, the error probability $\mathbb{P}_\mu[f(\mathbf{X}) \neq Y]$ is estimated by the *test error* $L(f, D_{n-m})$.

3.2.1 Model Feature Selection

Several machine learning models to classify BTS issues could be built. As in all uses of machine learning techniques, we are interested in selecting the most parsimonious model, *i.e.*, including the smallest possible subsets of characteristics (features) describing a BTS issue, but still having an acceptable accuracy. To account for both false positives and false negatives, we quantify accuracy via standard information retrieval measures: precision and recall [12].

Among all the available features (terms and other BTS fields) that could be used to describe a BTS issue, techniques such as backward elimination [20] can be applied to select a subset of the characteristics leading to the most accurate classifier. The cost of selecting variables is very high and can only be performed in off-line approaches as the ones used in this paper. Furthermore, multiple runs may be needed with different set of independent variable and different variable selection strategies in a trial-and-error approach.

In the following, we provide a short description of the algorithms used to build classifiers for BTS issues: decision trees, naive Bayes classifiers, and logistic regression.

3.2.2 Decision Tree

A decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers and terminal nodes contain one of the classification labels from the set C.

The decision making process starts at the root of the tree. Given an input vector $\mathbf{x}$ $(a_1, \ldots, a_d)$, the questions in the internal nodes are answered and the corresponding edges are followed. The label c of $\mathbf{x}$ is determined when a leaf is reached.

In our case, the leaf node are labeled with either 0 or 1 to indicate whether an issue is a bug or not. The internal node contains question regarding the values of various fields from the issue, for example whether the word "critical" appears in the text describing the entry or if the entry was tagged as "Enhancement".

In this paper we applied the Alternating Decision trees, or AD trees; AD trees are an extension of traditional classification trees that rely on boosting to produce robust classifiers [26].

3.2.3 Naive Bayes Classifier

A Bayesian classifier is a simple classification technique that classifies a d-dimensional observation $\mathbf{x}_i$ by determining its most probable class c computed as:

$$c \;=\; \arg\max_{c_k} p(c_k | a_1, \ldots, a_d),$$

where c_k ranges over the set of classes in C and the observation $\mathbf{x}_i$ is written as a generic attribute vector. By using *the rule of Bayes*, the probability $p(c_k | a_1, \ldots, a_d)$ called probability *a posteriori*, is rewritten as:

$$\frac{p(a_1, \ldots, a_d | c_k)}{\sum_{h=1}^{q} p(a_1, \ldots, a_d | c_h) p(c_h)} p(c_k).$$

The classifier structure is drastically simplified under the assumption that, given a class c_k, all attributes are conditionally independent. Under this assumption the following common form of *a posteriori* probability is obtained:

$$p(c_k|a_1,\ldots,a_d) = \frac{\prod_{j=1}^{d} p(a_j|c_k)}{\sum_{h=1}^{q}\prod_{j=1}^{d} p(a_j|c_h)p(c_h)} p(c_k). \quad (2)$$

When the independence assumption is made, the classifier is called naive Bayes classifier. The $p(c_k)$ marginal probability [9] (or base probability [8]) is the probability that a member of a class c_k will be observed. The $p(a_j|c_k)$ prior conditional probability is the probability that the j^{th} attribute assumes a particular value a_j given the class c_k. These two prior probabilities determine the structure of the naive Bayes classifier. They are learned, *i.e.*, estimated, on a training set when building the classifier.

3.2.4 Logistic Regression

In a logistic regression classifier, the dependent variable is commonly a dichotomous variable and, thus, C assumes only two values $\{0, 1\}$; The multivariate logistic regression classifier is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}} \quad (3)$$

where X_i are the characteristics describing the modeled phenomenon, and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In our problem, variable X_i will be words describing the issue. Thus, the closer the value is to 1, the higher is the probability that the bug tracking issue will describe a bug. To use the model as a classifier, a threshold is chosen. For example, if the threshold is equal to 0.5, an issue is considered to be a corrective maintenance request if $\pi > 0.5$.

4 Description and Oracle

Supervised learning techniques require a labeled corpus, a set of tagged BTS issues acting as the Oracle for the machine learning techniques. These issues are processed to extract characteristics also called *features*, used as independent variables by the various supervised techniques. This section defines the empirical study we performed, the addressed research questions, the oracle construction, the extraction of the features, and the automatic classification of the manually-indexed issues.

4.1 Description

The description of the study follows the Goal-Question-Metric paradigm [4]. The *goal* of this empirical study is to investigate how the text contained in issues posted on BTS can be used to classify such issues using machine learning techniques. The *quality focus* is achieving a high percentage of correctly classified issues for the two classes of issues that we consider: "bug", "non-bug". The *perspective* is both of researchers, who often use bug tracking information to build models (assuming that all issues are related to bugs and using simple pattern matching for the classification) and of project managers, who want to quickly distinguish particular kinds of issues. The *context* of this study is composed of three large open-source systems: Eclipse, Mozilla and JBoss.

This study aims at answering the research questions defined in the introduction:

- *RQ1: Issue classification.* To what extent the information contained in issues posted on bug tracking systems can be used to classify such issues, distinguishing bugs (*i.e.*, corrective maintenance) from other activities (*e.g.*, enhancement, refactoring)?
- *RQ2: Discriminating terms.* What are the terms/fields that machine learning models use to discern bugs from other issues?
- *RQ3: Comparison with grep.* Do machine learning approaches perform better than *grep* and regular expression matching in general, approaches often used to analyze CVS/SVN logs and classify commits between bugs and other activities?

4.2 Objects

We perform our study using three well-known, industrial-strength, open-source systems.

Eclipse is an open-source integrated development environment. It is a platform used both in open-source communities and in industry.

Systems	Bug	Non bugs	Others
Mozilla	270	209	121
Eclipse	194	382	24
JBoss	345	99	156

Table 1: Final classification over the 1,800 BTS issues for Mozilla, Eclipse and JBoss.

Eclipse is mostly written in Java, with C/C++ code used mainly for widget toolkits. Eclipse CVS and bug repositories were mirrored locally at the end of 2006. We extracted all bugs and selected 10,386 bugs, those tagged as either "Verified" or "Resolved", *i.e.*, bugs for which a resolution is known.

The Mozilla suite is an open-source suite implementing the Web browser and other tools such as mailers and newsreaders. It was ported on almost all software and hardware platforms. It is developed mostly in C++, with C code accounting for only a small fraction of the system. As for Eclipse, we are interested in the 92,858 bugs that are tagged as "Verified" or "Resolved".

JBoss is an enterprise-application platform and web-service application stack to develop, deploy, and manage Java service-oriented enterprise applications. It supports replication, transaction, caching, messaging, and clustering. It extends Java EE (J2EE 1.4) via features such as EJB3.0, Java Persistence API 1.0, Servlet 2.5, and so on. It is almost entirely developed in Java and XML plus shell scripts and batch files. As for Mozilla and Eclipse, we concentrate our effort on bugs for which a resolution was known. JBoss bugs are store in Jira. We use a RSS feeder to extract the 3,207 issues classified as "Resolved".

We select issues with the "Resolved" or "Closed" status to avoid duplicated bugs, rejected issues, or issues awaiting triage.

4.3 Building of the oracle

We classified, first manually, and then automatically, issues from the BTS of the three systems. It is clearly infeasible to manually classify each bug as either corrective maintenance or not because of the orders of magnitude of the numbers of retained bugs.

Therefore, we randomly sample and manually classify 600 issues for each system. Overall, 1,800 distinct issues are sampled. We organize the issues in bundles of 150 entries each. For every subset, we ask three software engineers to classify the issues manually. They are asked to state if the issues are a corrective maintenance (bugs) or a non-corrective maintenance (enhancement, refactoring, re-documentation, or other, *i.e.*, non bug). The classifications go through a simple majority vote and a decision on the status of each issue is made. An entry is considered a corrective maintenance if at least two out of three engineers classified it as a corrective maintenance (hereby referred to as "bug"). Otherwise the entry is considered as a non-corrective maintenance (hereby referred to as "non bug"). The obtained classifications are reported in Table 1

Manually classifying the issue is often a non trivial task: terms like "defect", "error", "bug" are not only used to describe corrective maintenance but also other form of maintenance. Indeed, as reported in Table 2, the dictionaries extracted from bug and non-bug issues share a non negligible number of words; they have non-null intersection. Thus, the presence of terms like "bug" or "defect" does not suffice to classify an issue as a bug, even if these terms are more frequent in corrective maintenance.

Table 1 suggests that Mozilla, Eclipse, and JBoss BTS contain a large fraction of non-bug issues. Notice the last column of the table: these are BTS issues that have nothing to do with bug fixing or evolution. For example, BTS issues in which a user complains of a problem related to his version of an operating system library unsupported by the application, requests an obsolete release, requests bug fixing of a component not belonging to the system, requests for write access to SVN/CVS repository, configuration help, and so on.

This labeled corpus is used to build explanatory as well as predictive models. These models could be then subsequently used to predict whether a new issue is a bug (thus, asking for the developers' attention) or something else (thus to be considered by the developers or the managers).

	Stemmed Dictionary Size		
Systems	Corrective Maintenance	Non Corrective Maintenance	Intersection
Mozilla	2,216	1,718	993
Eclipse	673	892	392
JBoss	3,575	1,386	1,065

Table 2: Characteristics of the dictionaries for Mozilla, Eclipse, and JBoss manually-classified issues.

4.3.1 Feature Extraction

Mozilla and Eclipse BTS issues are downloaded as HTML files while JBoss issues as XML documents. The downloading of the issues is performed using a Perl script wrapping the *wget* command. Feature extraction is also performed using another Perl script.

To apply machine learning techniques, we need to preprocess the HTML/XML texts of the issues. When extracting text from BTS issues, we consider and distinguish different linguistic features, namely: the title, the description, and the discussion following the description. In Bugzilla and Jira, these features have minor syntactic variation, thus two different parsers are needed to extract the textual part of title, description, and discussion.

Linguistic features undergo the standard processing, *i.e.*, text filtering, stemming, and indexing [12]. We do not apply stopping because it removes common English term such as "should", "might", "not". These terms are irrelevant in most general purpose information retrieval systems but may be important in our study. The semantic of a sentence "This is not a bug" is completely lost if the standard English stop-words are removed because the result is "This is bug". Filtering was limited to punctuation removal [12] and some specific transformations, such as splitting paths, e-mail addresses, and camel-case identifiers (*i.e.*, to divide "MyUser_account" into "My", "User", and "account"). Filtering is followed by a stemming phase, aimed at removing plural, identifying the infinitive of verbs, etc. using the Porter stemmer [6] from the *lsa* package of the statistical environment R[5].

A vector space is then built using the linguistic data [12]. Each BTS issue is mapped into a vector via the dictionaries built with filtered and stemmed data. Each vector element contains the raw frequency of a term in the document (*i.e.*, in the BTS issue). In previous work using linguistic data, good results were obtained using the *tf-idf* indexing instead of the raw frequency, because the use of the inverse document frequency (*idf*) penalizes terms appearing in too many documents (not discriminating). However, this is not so in our study: terms such as "failure", "crash", or "should" actually appear in many documents but, as we discover, constitute interesting *features* that guide the classification techniques to distinguish bugs from non-bugs.

Finally, each indexed BTS issue is augmented with its class $\{0, 1\}$, *i.e.*, $\{non\ bug,\ bug\}$. This column is used by the machine learning techniques during the training phase.

The best classification results on the three systems are achieved with the sole issue description; therefore, due to space limitation, we limit ourselves to report only those results in Section 5.

4.4 Automatic Classification

The automatic classification of BTS issues is performed using the *Weka* tool[6], in particular using the symmetrical uncertainty attribute selector, the standard probabilistic naive Bayes classifier, the alternating decision tree (ADTree), and the linear logistic regression classifier.

We first use a feature selection algorithm to select a subset of the available features with which to perform the automatic classification. Then, each automatic classifier is trained on a set of BTS issues and its performance is evaluated using *cross validation* [22]. In particular, we use a 10-fold cross validation, *i.e.*, dividing each set of issues in 10 sets, training the classifier on 9 of them, classifying the remaining set to test accuracy, and repeating the process to classify all the 10 sets.

[5] *http://www.r-project.org*

[6] *www.cs.waikato.ac.nz/ml/weka/*

5 Results and Discussions

This section reports the results of automatic classification of issues and of the evaluation of the performances of the obtained classifiers, answering the research questions formulated in Section 4.

5.1 RQ1: Issue Classification

Tables 3, 4 and 5 report the confusion matrices for Mozilla, Eclipse, and JBoss respectively. They are calculated from the automatic classification of issues using the different techniques, and selecting a different number of features, as described in Section 3.2.1. The matrices report correct classifications on the main diagonal (while the other two cells report wrong classifications) as well as precision and recall for both classes, and the percentage of correct decision (in bold).

For Mozilla (see Table 3), we note that increasing the number of features from 20 to 50 helps in increasing the recall of non-bug issues and the precision of bug issues, while decreasing the recall for bug issues, for ADTree and logistic regression; for ADtree, it also increases the precision for non-bug issues while for logistic regression, it decreases the precision from 73% to 71%.

The naive Bayes classifier only exhibits a limited improvement when increasing the number of features, with an increase of the recall from 49% to 65% and of the precision of non-bug issues from 57% from 64%.

Overall, considering precision and recall for both classes, regression logistics appears to perform better than other classifiers, although differences with the naive Bayes classifier are limited. Interestingly, the logistic regression with 50 features has an overall correct decision rate of 77% much higher of any random or constant classifies and substantially better of naive Bayes or AD trees.

No further improvements were found when increasing the number of selected features above 50.

For Eclipse (see Table 4), the improvement obtained when increasing the number of features is very limited if any. The performance of the three classifiers is similar (with the exception of the ADTree that has a lower recall for bugs: 52% with 20 features and 49% with 50 features). Again, the logistic regression performs slightly better than other techniques not only in terms of precision and recall but also as percentage of correct decisions with a 18% error rate.

For JBoss (see Table 5), we consider a larger number of features than for Mozilla and Eclipse, 50 and 100, because we obtain poor results with 20 features. Overall, we note that (1) for all three classifiers, there is a slight improvement when using 100 features instead of 50 and (2) that the logistic regression again out-performs other classifiers, although the recall for non-bug issues is very low with 29%, still it achieves an 82% correct decision.

The data in Tables 3, 4 and 5 show that the classifiers out-perform the random and constant classifiers (that would always classify an issue as bug or as non-bug), although performances of the automatic classifiers vary. For example, a constant classifier on Mozilla would always answer "It is a bug" and, thus, would have a 100% recall and a precision of 56% on bugs at the price of a null recall on non-corrective maintenance; overall it has 56% correct decision.

We therefore conclude that the information contained in issues posted on bug tracking systems can be indeed used to classify such issues, distinguishing bugs from other activities, with a precision between 64% and 98% and a recall between 33% and 97%, and an error rate between 18% and 23%.

5.2 RQ2: Discriminating Terms

In addition to showing that the information contained in the issue is enough to classify them as bug or non-bug, we study the features that are used to perform the classification. This study is possible for the decision tree, for which the tree itself can be visualized, and for the logistic regression, for which the variables and their related coefficients can be more easily studied. Moreover, the number of features *really* used by these two classifiers is smaller than the number of selected features because both these classifiers perform a further selection over the subset of all provided features.

Selected Features		Naive Bayes			ADTree			Logistic Regression		
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
20	Bug	133	137	49%	251	19	93%	240	30	89%
	Non-bug	28	181	87%	142	67	32%	127	82	39%
	Prec.	83%	57%	**66%**	64%	78%	**66%**	65%	73%	**67%**
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
50	Bug	175	95	65%	162	108	60%	205	65	76%
	Non-bug	41	168	80%	49	160	77%	46	163	78%
	Prec.	81%	64%	**72%**	77%	60%	**68%**	82%	71%	**77%**

Table 3: Mozilla: automatic classification confusion matrices (in bold percentage of correct decisions).

Selected Features		Naive Bayes			ADTree			Logistic Regression		
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
20	Bug	110	84	57%	101	93	52%	114	80	59%
	Non-bug	30	352	92%	27	355	93%	32	350	92%
	Prec.	79%	81%	**80%**	79%	79%	**79%**	78%	81%	**81%**
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
50	Bug	117	77	60%	96	98	49%	120	74	62%
	Non-bug	33	349	91%	29	353	92%	29	353	92%
	Prec.	78%	82%	**81%**	77%	78%	**78%**	80%	83%	**82%**

Table 4: Eclipse: automatic classification confusion matrices (in bold percentage of correct decisions).

Figure 1 shows an example of an ADtree for Mozilla. Decision points with a positive coefficient indicate that, if the expression on the right side is true, the decision is leaning towards classifying the issue as a "bug". Negative coefficients indicate a decision towards classifying the issue as "non-bug". It is worth to be noted that the terms represented in the tree are *stem* of words contained in the BTS issues.

We note that terms such as "crash", "critic", "broken", "when" (often used when one wants to reproduce an issue) lead to classifying the issue as a "bug", while terms such as "should", "implement", "support" cause a classification as "non-bug" (mostly a request for enhancement or for a new feature).

Moreover, when Bugzilla "Severity" field is used—field that should actually be used to distinguish bugs from non-bugs—the "Enhancement" value indicates a possible classification as non-bug, while "major" as bug.

Figure 2 shows an example of ADTree for Eclipse (negative coefficients mean non-bug). Again, when the "Severity" field is used, "Enhancement" leads towards a non-bug classification. Terms, such as "failur(e)", "fail", "npe" (null-pointer exception), "except(ion)", "error", lead towards a bug classification.

Figure 3 shows the logistic regression coefficients obtained for Eclipse. Positive coefficients lead the classification tend towards a non-bug classification, while negative coefficients towards a bug classification. Again, the "Severity" field plays an important role: "Enhancement" indicates a non-bug issue while all other values a possible non-bug.

Interestingly, we note that the value "trivial" was used for some issues in Bugzilla "Severity" tag for bugs. Terms having a high influence for the "bug" classification are, for example, "except(ion)", "fail", "npe" (null-pointer exception), "error", "correct", "termin(ation)", "invalid". Terms such as "provid(e)", "add", possibly indicate a non-bug issue.

Figure 4 shows logistic regression coefficients for JBoss. Negative coefficients lead the classifier towards a non-bug classification. Although we still have terms such as "pointer" and "incorrect" among the terms used for the bug classification and terms such as "should" for the non-bug classification, the interpretation is less intuitive.

Selected Features		Naive Bayes			ADTree			Logistic Regression		
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
50	Bug	112	232	33%	319	25	92%	335	9	97%
	Non-bug	4	95	96%	86	13	13%	78	21	21%
	Prec.	97%	29%	**46%**	79%	34%	**75%**	81%	70%	**80%**
		Predicted			Predicted			Predicted		
		Bug	Non-bug	Rec.	Bug	Non-bug	Rec.	Bug	Non-bug	Rec.
100	Bug	145	199	42%	315	29	92%	333	11	97%
	Non-bug	3	96	97%	76	23	23%	70	29	29%
	Prec.	98%	32%	**54%**	80%	44%	**76%**	83%	72%	**82%**

Table 5: JBoss: automatic classification confusion matrices (in bold percentage of correct decisions).

```
|  (1)crash < 0.5: 0.056
|  |  (9)broken < 0.5: 0.039
|  |  (9)broken >= 0.5: -0.9
|  (1)crash >= 0.5: -1.639
|  (2)SEV = _enhancement: 1.334
|  (2)SEV != _enhancement: -0.072
|  |  (4)when < 0.5: 0.057
|  |  |  (5)SEV = _major: -0.604
|  |  |  (5)SEV != _major: 0.053
|  |  (4)when >= 0.5: -0.636
|  (3)support < 0.5: -0.03
|  |  (7)should < 0.5: -0.064
|  |  |  (8)from < 0.5: 0.032
|  |  |  (8)from >= 0.5: -0.825
|  |  |  (10)implement < 0.5: -0.046
|  |  |  (10)implement >= 0.5: 1.071
|  |  (7)should >= 0.5: 0.462
|  (3)support >= 0.5: 1.421
|  (6)critic < 0.5: 0.039
|  (6)critic >= 0.5: -0.759
```

Figure 1: Mozilla: Example of ADtree

```
|  (1)SEV = enhancement: -1.587
|  (1)SEV != enhancement: 0.291
|  |  (3)not < 0.5: -0.086
|  |  |  (7)SEV = normal: -0.08
|  |  |  (7)SEV != normal: 0.309
|  |  |  (10)failur < 0.5: 0.034
|  |  |  (10)failur >= 0.5: 0.775
|  |  (3)not >= 0.5: 0.64
|  (2)except < 0.5: -0.063
|  |  (4)fail < 0.5: -0.07
|  |  |  (8)npe < 0.5: -0.042
|  |  |  |  (9)doe < 0.5: -0.034
|  |  |  |  (9)doe >= 0.5: 0.577
|  |  |  (8)npe >= 0.5: 1.06
|  |  (4)fail >= 0.5: 1.428
|  (2)except >= 0.5: 1.58
|  (5)error < 0.5: -0.024
|  |  (6)when < 0.5: -0.04
|  |  (6)when >= 0.5: 0.631
|  (5)error >= 0.5: 0.891
```

Figure 2: Eclipse: ADtree example

Therefore, we conclude that certain terms and fields lead to more discriminating classifiers between "bug" and "non-bug" issues.

5.3 RQ3: Comparison with *grep*

To assess the usefulness of the machine-learning classifiers, it is useful to compare their performance with those of the simplest classifier that developers would have used: string and regular expression matching, e.g. using the Unix utility *grep*. For example, developers would search for files containing the term "bug" or synonyms such as "defect" or "problem".

Several combinations of conjunctions or disjunctions with synonyms and abbreviations of the term bug are possible. We apply the regular expression introduced by [21] and [11] to classify bugs and recovery traceability links between BTS and CVS or SVN repositories. More details on this technique can be found in [3].

More precisely, we classify issues my means of the following *grep* regular expression to maximize retrieval:

$$\backslash bfix| \backslash bbug| \backslash bproblem| \backslash bdefect| \backslash bpatch$$

Each hit on the filtered textual information of the 1,800 manually-classified bugs was considered as a detected bug; multiple hits on the same issues were not counted. This regular expression is a minor variant of [21, 11, 3]. The difference is the term $\backslash b$, a word boundary (*e.g.*,

```
[SEV=trivial] * -0.26 +
[SEV=enhancement] * 1.69 +
[SEV=blocker] * -0.42 +
[SEV=major] * -0.45 +
[SEV=critical] * -0.21 +
[except] * -2.36 +
[fail] * -2.42 +
[doe] * -0.71 +
[not] * -0.58 +
[when] * -0.42 +
[error] * -0.92 +
[npe] * -1.97 +
[add] * 0.4  +
[correct] * -1.7 +
[provid] * 0.38 +
[termin] * -1.98 +
[other] * -1.63 +
[bad] * -1.75 +
[invalid] * -1.62 +
[record] * -1.41 +
[renam] * -1.07 +
[work] * -0.27 +
[failur] * -1.03
```

Figure 3: Eclipse: Logistic Regression Coefficients

white space or a parenthesis) that we use to decrease the number of false positive (e.g., to avoid matching "preFIX").

The data in Tables 6,7, 8 shows the accuracy of using the regular expression with respect to the manual classification, *i.e.*, bug and non-bug data from Table 1. Precisions and recalls are not very high, if compared to the results obtained using other classifiers and shown in Tables 3, 4 and 5. In addition, since a simple string-matching is performed, it is impossible to automatically distinguish a meaningful matching from a false positive matching.

		Predicted		Recall
		Bug	Non Bug	
Issues	Bug	23	247	9%
	Non bug	12	197	94%
Precision		66%	44%	**46%**

Table 6: Mozilla *grep* confusion matrix for manually classified bugs (in bold percentage of correct decisions).

As reported in the tables, *grep* achieves at most 67% correct decisions (*i.e.*, on Eclipse) but with a bug recall of only 6%. We can conclude that a naive approach, using *grep*, is no match to the previous classifiers.

```
[jboss] * 0.04 +
[sentenc] * -1.13 +
[lang] * 0.07 +
[session] * 0.15 +
[pointer] * 0.32 +
[except] * 0.07 +
[jbpm] * -0.12 +
[should] * -0.31 +
[fine] * 0.53 +
[default] * 0.11 +
[excess] * -1.52 +
[recoveri] * -0.5 +
[cmp] * 0.12 +
[commit] * 0.14 +
[trigger] * -0.61 +
[kind] * -0.97 +
[concurr] * 0.34 +
[cnfe] * -0.6 +
[wonnekeys] * -1.16 +
[housekeep] * -0.91 +
[intend] * -1.15 +
[simpler] * -1.22 +
[reload] * -0.79 +
[easier] * -1.19 +
[especi] * -1.31 +
[dom] * -0.39 +
[tell] * -1.05 +
[transfer] * -0.47 +
[mistak] * -0.98 +
[revers] * -0.59 +
[then] * 0.25 +
[multipl] * 0.47 +
[incorrect] * 0.32 +
[modifi] * 0.52 +
[ant] * 0.33
```

Figure 4: JBoss: Logistic Regression Coefficients

5.4 Threats to the Validity

This section discusses threats to validity that can affect our study, following the guidelines for case study research provided by Yin [27].

Threats to *construct validity* concern the relationship between the theory and the observation. In this case the threat can be mainly due to the use of incorrect operational measures concerning the investigated phenomenon. In our study we used all sources of information

		Predicted		Recall
		Bug	Non Bug	
Issues	Bug	11	183	6%
	Non bug	8	374	98%
Precision		58%	67%	**67%**

Table 7: Eclipse *grep* confusion matrix for manually classified bugs (in bold percentage of correct decisions).

		Predicted		Recall
		Bug	Non Bug	
Issues	Bug	106	239	31%
	Non bug	31	68	69%
Precision		77%	22%	**39%**

Table 8: JBoss *grep* confusion matrix for manually classified bugs (in bold percentage of correct decisions).

BTS contributors use to report issues, namely structured fields and free text, for the latter considering both title, description and discussion. However, we found that the discussion was not useful for our classification purposes. Therefore, there is no threat to the construct validity of our study.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, this kind of threats can be due to a possible level of subjectiveness caused by the manual construction of oracles, and to the bias that can be introduced by the manual classifiers if they are aware of the classification algorithm to be used. We attempted to avoid any bias in the building of the oracle and of the classifiers by first classifying each issues manually without making any choice on the classifiers to be used. The manual classification was performed independently by each engineer and then combined using the majority voting technique.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to the classification of BTS issues coming from Mozilla, Eclipse and JBoss. Although the approach is perfectly applicable to other systems, we do not know whether the same results will be obtained. In addition, we have built different classifiers for each system, where the discriminating features and terms were different. Finally, the classification was limited to a subset of Mozilla, Eclipse and JBoss BTS issues, since is not feasible to verify by hands about 100,000 issues. We downloaded 92,858, 10,386 and 3,207 issues from BTS of Mozilla, Eclipse and JBoss respectively. In a previous work [3] we randomly selected and manually classified a sample of 600 Mozilla issues out of the 35,000. These previously classified issues are part of the study. Such a sample size was sufficient to ensure a confidence level of 95% and a confidence interval of $\pm$ 10% for precision and recall in the context of the study [3] i.e., 35,000 issues. Although, the number of manually classified bugs for Eclipse and JBoss ensures an even higher confidence level and a smaller confidence interval, we cannot guarantee the same for Mozilla results. In essence, it is not impossible that better results can be achieved on Mozilla; results closer to those of Eclipse and JBoss.

6 Conclusion

This paper shows that linguistic information contained in BTS entries is sufficient to automatically distinguish corrective maintenance from other activities. This is relevant in that it opens the possibility of building automatic routing systems, i.e., systems that automatically classify submitted tickets and route them to the maintenance team (bugs) or to team leader (enhancement requests and other issues). We also report on our experience in manually classifying 1,800 issues from the BTS of three large open-source systems: Mozilla, Eclipse, and JBoss. We show that, using a simple voting mechanism, it is possible to classify this large number of issues, distinguishing between "bugs" (related to corrective maintenance) and "non-bug" issues (related to other maintenance activities, documentation, and so on).

Using the manually-classified issues as an oracle, we build classifiers for each systems using three supervised machine learning techniques: alternating decision trees (ADtree)s, naive Bayes classifiers, and logistic regression. With the classifiers, we answered three research question and concluded that:

- the information contained in issues posted on bug tracking systems can be indeed used to classify such issues, distinguishing

bugs from other activities, with a precision between 64% and 98% and a recall between 33% and 97% and a correct decision rate as high as 82%;

- certain terms and fields lead to more discriminating classifiers between "bug" and "non-bug" issues;
- a naive approach, using *grep*, is no match for the classifiers built using our oracle.

In addition, we can report that, out of the 1,800 manually-classified issues, less than half are related to corrective maintenance. Therefore, bug tracking systems, in open-source development, have a far more complex use than simple bookkeeping of corrective maintenance. Also, study based on BTS issues should carefully consider what issues are used to build their predictive models.

Future work includes studying the relation with bug, bug fixing and design patterns, and modeling bug fixing induced by previous corrective maintenance interventions.

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658) and by G. Antoniol NSERC Discovery Grant.

About the Authors

Giuliano Antoniol received his degree in electronic engineering from the Universita' di Padova in 1982. In 2004 he received his PhD in Electrical Engineering at the Ecole Polytechnique de Montreal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research Chair Tier I in Software Change and Evolution.

Giuliano Antoniol published more than 100 papers in journals and international conferences. He served as a member of the Program Committee of international conferences and workshops such as the International Conference on Software Maintenance, the International Conference on Program Comprehension, the International Symposium on Software Metrics. He is presently member of the Editorial Board of the Journal Software Testing Verification & Reliability, the Journal Information and Software Technology, the Journal of Empirical Software Engineering and the Journal of Software Quality.

He is currently Full Professor at the Ecole Polytechnique de Montreal, where he works in the area of software evolution, software traceability, search based software engineering and software maintenance

Kamel Ayari is a Ph.D. candidate at the Ecole Polytechnique de Montreal, he joined Dr. Giuliano Antoniol's team in 2006 and is interested in working on search based software engineering and software maintenance.

Massimiliano Di Penta is assistant professor at the University of Sannio in Benevento, Italy and researcher leader at the Research Centre On Software Technology (RCOST). He received his PhD in Computer Engineering in 2003 and his laureate degree in Computer Engineering in 1999. His main research interests include software maintenance, reverse engineering, empirical software engineering and service-oriented software engineering. He is author of over 90 papers published on referred journals, conferences and workshops. He is program co-chair of the 14th Working Conference on Reverse Engineering (WCRE 2007), of the 9th International Symposium on Web Site Evolution (WSE 2007), of the 9th IEEE International Workshop on Principles of Software Evolution (IWPSE 2007) and steering committee member of CSMR and STEP. He has been program co-chair of WCRE 2006, SCAM 2006 and STEP 2005. He serves and has served program and organizing committees of conferences and workshops such as CSMR, GECCO, ICSM, IWPC, SCAM, SEKE, WCRE, and WSE. He is member of the IEEE, the IEEE Computer Society and of the ACM.

Foutse Khomh is a Ph.D. candidate in Computer Science at the University of Montreal (Canada). The primary focus of his Ph.D. thesis is to develop techniques and tools to assess the quality of the design and implementation of large software systems and to ensure the traceability of design choices during evolution. The final result of the thesis will be a qual-

ity model that takes into account the quality of the design of large systems and thus that provide both a more detailed and high-level view on quality. He received a Master's degree in Software Engineering from the National Advanced School of Engineering (Cameroon) and a D.E.A (Master's degree) in Mathematics from the University of Yaounde I (Cameroon). He also has experience as Software Engineer at different companies doing system design and project management.

Yann-Gal Guhneuc is assistant professor at the Department of Computing Science and Operations Research of University of Montreal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from cole des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals.

References

[1] *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[2] Ethem Aplaydin. *Introduction to Machine Learning*. MIT Press, 2004.

[3] Kamel Ayari, Peyman Meshkinfam, Giulio Antoniol, and Massimiliano Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *CASCON*, Toronto, CA, Oct 23-25 2007.

[4] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[5] L. C. Briand, S. Morasca, and V. Basili. Measuring and assesing maintainability at the end of high level design. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 88–97, Montreal, 1993.

[6] S.E. Robertson C.J. van Rijsbergen and M.F. Porter. *New models in probabilistic information retrieval*. London: British Library, Research and Development Report, no. 5587, 1980.

[7] Rumelhart D. E., Hinton G. E.and, and Williams R. J. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[8] ElkanC. Naive bayesian learning. Technical report, Department of Computer Science Harvard University, 1997.

[9] FentonN. and NeilM. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[10] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution*, 16(6):115–141, 2004.

[11] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam Netherlands, September 2003. IEEE Computer Society Press.

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

[13] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of IEEE International Conference*

on Software Maintenance, pages 190–197, 1998.

[14] Daniel M. German. An empirical study of fine-grained software modifications. *Journal of Empirical Software Engineering*, 2005.

[15] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.

[16] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[17] N. Kurishima, H. Oikawa, J. Nakamura, K. Amari, M. Fujioka, and K. D. Denwa. Quantitative analysis of error in telecomunications software. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 190–198, Victoria, 1994.

[18] Tom Mitchell. *Machine Learning*. MIT Press, 1997.

[19] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[20] J.O. Rawlings, S. G. Pandula, and D. A. Dickey. *Applied Regression Analysis a Research Tool*. Springer Texts in Statistics. New York: Springer-Verlag, second edition edition, 1998.

[21] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories MSR 2005 Saint Louis Missouri USA*, May 17 2005.

[22] M. Stone. Cross-validatory choice and assesment of statistical predictions (with discussion). *Journal of the Royal Statistical Society B*, 36:111–147, 1974.

[23] Marek Vokavc. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Software Eng.*, 30:904–917, 2004.

[24] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 461–470, New York, NY, USA, 2008. ACM.

[25] Peter Weissgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 International Workshop on Mining Software Repositories MSR 2006 Shanghai China May 22-23 2006*, pages 112–118, 2006.

[26] Ian Witten and Eibe Frank. *Data Mining Practical Machine Learning Tools and Techniques - Second Edition*. Elsevier, 2005.

[27] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[28] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

[29] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, pages 563–572, 2004.