BUGMINER: Software Reliability Analysis Via Data Mining of Bug Reports

Leon Wu Boyi Xie Gail Kaiser Rebecca Passonneau
Department of Computer Science
Columbia University
New York, NY 10027 USA
{leon,xie,kaiser,becky}@cs.columbia.edu

Abstract

Software bugs reported by human users and automatic error reporting software are often stored in some bug tracking tools (e.g., Bugzilla and Debbugs). These accumulated bug reports may contain valuable information that could be used to improve the quality of the bug reporting, reduce the quality assurance effort and cost, analyze software reliability, and predict future bug report trend. In this paper, we present BUGMINER*, a tool that is able to derive useful information from historic bug report database using data mining, use these information to do completion check and redundancy check on a new or given bug report, and to estimate the bug report trend using statistical analysis. Our empirical studies of the tool using several real-world bug report repositories show that it is effective, easy to implement, and has relatively high accuracy despite low quality data.*

1 Introduction

Finding and fixing the faults in software is an indispensable while time-consuming quality assurance task in software development. Our definition of *fault* is a programming error that leads to an erroneous result in some programs during execution. A *software bug* is the common term used to describe a fault, error, flaw, mistake, or failure in a program that produces an incorrect or unexpected result, or causes it to behave in unintended ways. When a software bug is identified, it is often reported and recorded into a bug report database using some bug tracking tools so that further analysis or fix can be performed, possibly by a developer or tester. For some real-world software, their bug report databases have accumulated a large amount of historic bug reports. For example, as of February 2011, Debbugs, *i.e.*, Debian bug tracking system, has accumulated 615,000 bug reports [6, 5].

These accumulated bug reports may contain valuable information that could be used to improve the quality of the bug reporting, reduce the cost of quality assurance, analyze software reliability, and predict future bug report trend. One of the challenges in bug reporting is that the bug reports are often incomplete (*e.g.*, missing data fields such as product version or operating system details). Another challenge is that there are often many duplicate bug reports for the same bug. Software developers or testers normally have to review these redundant bug reports manually, which is time-consuming and cost inefficient.

We developed a tool named BUGMINER that is able to derive useful information from historic bug reports using data mining techniques, including machine learning (*e.g.*, SVM [16, 3]) and natural language processing, and use these information to do completion check through classification and redundancy check through similarity ranking on a new or given bug report. BUGMINER can also perform bug report trend analysis using Weibull distribution [14]. We implemented the tool and experimented it using three real-world bug report repositories including Apache Tomcat [1], Eclipse [7], and Linux Kernel [12]. Our experiments demonstrate that it is effective, easy to implement, and has relatively high accuracy despite low quality data.

The rest of the paper is organized as follows. In the following section, we give background information on bug reporting. In Section 3, we present the details of our approach, followed by our empirical studies in Section 4. Lastly, we compare related work in Section 5, before we conclude in Section 6.

2 Background on Bug Reporting

Bug tracking tools are often developed as a database-driven web application. The web interface allows multiple geographically distributed users to enter the bug reports simultaneously. The backend database stores the records for the reported bugs. Table 1 lists some main attributes (*i.e.*, data fields or columns) of a typical bug report for Apache Tomcat using Bugzilla [1, 2]. These attributes are meta in-

formation of the bug report. The field bug_id is an unique identifier for a distinct bug instance. A bug report is often modified by subsequent reviewers or developers who are trying to verify or fix the bug. Table 2 lists the additional commentary entries related to the same bug listed in Table 1. Each new entry (*i.e.*, new long_desc record) records the author name, entry date and time, and the free text description. The entry date and time for the first and last long_desc record, along with the first author's name, are also stored in the main attributes list of the same bug (*i.e.*, creation_ts, delta_ts, and reporter). There is no predefined limit on how many additional commentary entries a bug report can hold. Bug report datasets will be further explained in Section 4.2.

Table 1. Main attributes of a bug report

Attribute Name	Sample Value	Attribute Name	Sample Value
bug_id	48892	component	Connectors
creation_ts	2010-03-11 12:10:09 -0500	delta_ts	2010-12-14 14:30:22 -0500
short_desc	Use URIEncoding...	rep_platform	All
cclist_accessible	1	op_sys	All
classification_id	1	bug_status	NEW
classification	Unclassified	bug_severity	enhancement
product	Tomcat 7	priority	P2
reporter	reporter 1	assigned_to	dev

Table 2. Additional attributes

Attribute Name	long_desc 1	long_desc 2	long_desc 3
isprivate	0	0	0
who	reporter 1	reporter 2	reporter 3
bug_when	2010-03-11 12:10:09 -0500	2010-04-04 10:18:48 -0400	2010-12-14 14:30:22 -0500
thetext	Here is a ...	There are ...	For encoding ...

3 Approach

3.1 Architecture

Figure 1 illustrates the architecture of BUGMINER. There are two types of bug reporters: human users such as software developers, testers, and end users; automatic error reporting processes that run as a service on users' computers. The bug reporters generate new bug reports and enter the related information via the bug tracking tool's interface. The bug tracking tool then store the new bug report into the bug report database.

BUGMINER consists of three data mining and statistical processing engines: automatic completion check engine; automatic redundancy check engine; and bug report trend analysis engine. These three engines process the historic data stored in the bug report database and the new bug report coming in. The results from these engines are then directed to the bug tracking tool so that these results can be reviewed and stored. In the following subsections, we will describe each engine in detail.

3.2 Attributes and Feature Selection

BUGMINER analyzes bug report data based on two sets of attributes: 1) static meta information, and 2) bag-of-words (*i.e.*, a collection of distinct free text words) attributes. For each bug report in Bugzilla, users need to fill in a predefined set of bug information, as shown in Table 1. This set of attributes has two characteristics: 1) static: the list of fields is fixed for all types of software products, and those fields are available for all bug reports; 2) meta information: they describe the general information about the bug report but doesn't go to the details of the problem. Bug report analysis based solely on the static and meta information is very limited. In BUGMINER, we further include the free text data of a bug report in our analysis.

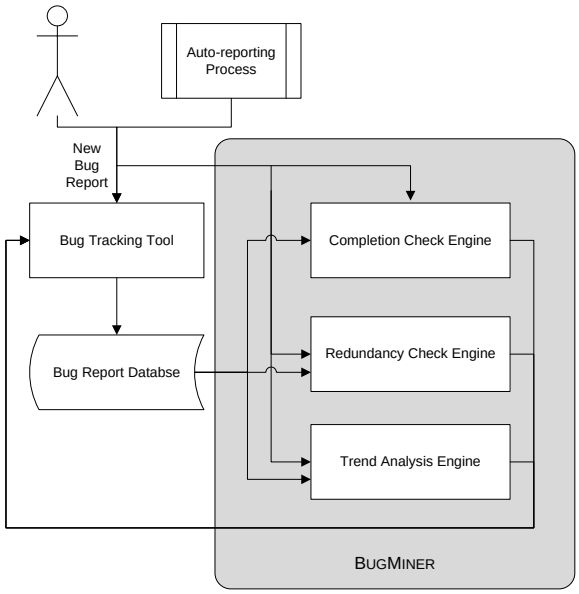

Figure 1. BUGMINER **architecture**

The free text data usually describes a bug scenario in natural language followed by some sample code. We represent the textual data as a bag-of-words. Each data instance is a high dimensional vector with words being attributes. The values of the attributes are Term Frequency-Inverse Document Frequency (TF-IDF) weight [13], which gives a higher weight on words that are frequent in certain data records but not too common across all records. Stemming, a process of reducing inflected (or sometimes derived) words to their stem or root form, is not necessary because the vocabulary usually doesn't have a variety of morphed forms, and imperfect stemming may bring in additional noisy content unnecessarily. Our feature selection also bases on inverse document frequency (IDF) and global term frequency. Words with a low IDF (*e.g.*, stopwords such as 'the' and 'a') are removed because they are too common and lack discriminative power. Words with a very low global term frequency are also removed because they are rare and their inclusion leads to a high dimensionality, which may cause "curse of dimensionality" problem in machine learning.

3.3 Automatic Completion Check Engine

When a bug report is filed, the bug information submitted are sometimes incomplete (*e.g.*, missing data fields). BUGMINER's automatic completion check engine derives these missing data fields through mining historic data and classification using the partially filled information. It reduces manual effort, keeps the bug report complete, and helps developers and testers to analyze the bug report more precisely.

3.3.1 Classification Using Support Vector Machine

Missing field autocompletion can be solved as a supervised learning problem. By training a classification model on existing data, we can predict the missing values. In BUGMINER, we use Support Vector Machines (SVM) as the classifier. SVM is a popular machine learning method because of its high performance. It formulates the classification modeling process as a quadratic minimization problem, and finds hyperplanes in a high dimensional space that separate data instances of different categories, while maximizing the margins between categories.

We first use a set of historic bug reports (*e.g.*, each one with n attributes) as training data to build a linear SVM model. For a new or given bug report with one missing data field a (*i.e.*, $n-1$ attributes filled and 1 attribute missing), we use the trained SVM model as a classifier and the filled $n-1$ attributes to predict the value of the missing a field for this bug report. In the case of multiple data fields are missing for a report (*e.g.*, $n-m$ attributes filled and m attributes missing), we use the SVM model and the $n-m$ filled attributes to predict the missing fields one by one.

3.4 Automatic Redundancy Check Engine

A common way of searching a bug report database to find out whether a new or given bug report already exists or not is to use keyword search, which normally uses keyword in combination with some wildcat characters such as '%' and '?' to construct database query string that can be executed on the database table. This kind of search based on keyword matching is often imprecise and may generate a large amount of useless or irrelevant results. The similarity ranking used by BUGMINER's automatic redundancy check engine is able to tell whether the new bug report is a duplicate or not more precisely. Furthermore, the similarity ranking can find out the most similar prior bug reports and sort them for the user.

3.4.1 Similarity Ranking Using Cosine Similarity

We represent bug report dataset in a vector space model (*i.e.*, term vector model), an algebraic model for representing text documents as vectors of identifiers, such as index terms [15]. Each bug report is a vector that consists of a list of feature values. As described in Section 3.2, BUGMINER uses two sets of features: 1) static meta information; 2) bag-of-words attributes with TF-IDF values.

We measure the similarity between two bug reports based on Cosine similarity, *i.e.*, the Cosine of the angle between the two vectors that represent these two bug reports, as shown in the following formula:

$$Distance_{COS}(a,b) = \frac{\sum_i a_i \times b_i}{\sqrt{\sum_i a_i^2} \times \sqrt{\sum_i b_i^2}},$$

where a and b represent two vectors. Its result equals 1 when the angle between two vectors is 0 (*i.e.*, two vectors are pointing in the same direction), and its result is less than 1 otherwise.

For a new or given bug report, we compute the Cosine similarity value (*i.e.*, csv) between this new bug report's vector and all the prior bug reports' vectors, and then rank the csv values in an descending order. The historic bug report with the highest csv value (*i.e.*, the closest one to 1) is the most similar prior record.

3.4.2 Similarity Ranking Using KL Divergence

In addition to Cosine similarity, we rank all prior bug reports based on their relevance to the new bug report using probability distribution. Kullback-Leibler (*i.e.*, KL) divergence [4, 13] is an effective relevance metric that assumes each data instance in a high dimensional feature space is characterized by a probability distribution. KL divergence measures the dissimilarity between two probability distributions, as shown in the following formula:

$$D_{KL}(a||b) = \sum_{t \in V} P(t|M_a) log \frac{P(t|M_a)}{P(t|M_b)},$$

where M_a and M_b represent the probability distributions for vector a and b respectively. V is the vocabulary of all terms and t is a term in V. KL divergence measures how bad the probability distribution M_a is at modeling M_b. Previous work [11] presents results suggesting that model comparison approach outperforms both query-likelihood and document-likelihood approaches. However, this metric is asymmetric, *i.e.*, $D_{KL}(a||b) \neq D_{KL}(b||a)$. In order to use it as a distance metric, we adopt a symmetrized KL divergence method for similarity ranking, which is defined as:

$$Distance_{KL}(a,b) = \frac{1}{2} D_{KL}(a||b) + \frac{1}{2} D_{KL}(b||a).$$

The result is symmetric and nonnegative. It equals 0 when two distributions are identical. It is bigger than 0 otherwise, and the larger the value the greater their dissimilarity.

For a new or given bug report, we compute the symmetrized KL divergence value (*i.e.*, kld) between this new bug report's vector and all the prior bug reports' vectors,

and then rank the kld values in an ascending order. The historic bug report with the lowest kld value (*i.e.*, the closest one to 0) is the most similar prior record.

3.4.3 Is the New Bug Report a Duplicate? What are the Similar Bugs Reported before?

We categorize a new or given bug report into one of the three categories according to the ranked csv and kdl values, along with the value ranges they fall into:

- If a prior report exists with $csv \geq c\_r_2$ and $kld \leq k\_r_1$, it is highly likely to be a duplicate (or repeat) of a prior report.
- If a prior report exists with $c\_r_1 < csv < c\_r_2$ or $k\_r_1 < kld < k\_r_2$, it has similar prior report.
- If all prior reports have $csv \leq c\_r_1$ and $kld \geq k\_r_2$, it does not have any similar prior report.

The value range parameters (*i.e.*, c_r_1, c_r_2, k_r_1, and k_r_2) can be determined based on heuristics obtained from experiments.

3.5 Bug Report Trend Analysis Engine

After major software releases, the number of software bugs tend to increase initially. As these bugs are fixed, the number of bugs gradually decreases, which resembles the "bathtub curve" in reliability engineering. The increase and decrease of the number of bugs normally lead to the similar trend of the number of bug reports. Weibull distribution can be used to model this kind of pattern and provide the basis for trend analysis.

3.5.1 Report Incidence Distribution

For the Weibull distribution, the incidence (*e.g.*, failure or bug report) density function $f(t)$ and cumulative incidence distribution function $F(t)$ are

$$f(t;\lambda,k) = \frac{k}{\lambda}(\frac{t}{\lambda})^{k-1}e^{-(t/\lambda)^k}, \quad t \geq 0,$$

$$F(t;\lambda,k) = 1 - e^{-(t/\lambda)^k}, \quad t \geq 0,$$

where $k > 0$ is the shape parameter and $\lambda > 0$ is the scale parameter of the distribution. The instantaneous incidence rate (or hazard function) when $t \geq 0$ can be derived as

$$h(t;\lambda,k) = \frac{f(t;\lambda,k)}{1 - F(t;\lambda,k)} = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1}.$$

A value of $k < 1$ indicates that the incidence rate decreases over time. A value of $k = 1$ indicates that the incidence rate is constant (*i.e.*, k/λ) over time. In this case, the Weibull distribution becomes an exponential distribution. A value of $k > 1$ indicates that the incidence rate increases with time.

3.5.2 Estimation of Coming Bug Report

We first use historic data to fit the Weibull function and derive the λ and k parameters. Then for any given time t, which is the number of weeks (or other chosen time units such as days or hours) after the starting date, the number of bug reports that may happen during that week can be estimated using the Weibull's density function $f(t)$. The result is an estimate of how many bug reports may happen during the t-th week after the starting event, *e.g.*, a new software release. Similarly, the instantaneous incidence rate can be estimated using the hazard function $h(t)$. These estimates give software developers or testers a baseline for designing the software testing and maintenance plan.

4 Empirical Studies

4.1 Implementation

We implemented BUGMINER in Java using some existing machine learning and statistical analysis tools, including Weka [18] and MATLAB [9].

4.2 Bug Report Datasets and Data Processing

We experiment BUGMINER on the bug report repositories of three real-world software applications (Apache Tomcat [1], Eclipse [7], and Linux Kernel [12]). Table 3 lists some statistics of these bug report repositories. For example, the Apache Tomcat dataset contains two product versions—Tomcat 3 and Tomcat 7. The OS is the operating system the software runs on. The components are the functional components of the software.

Table 3. Software and bug report datasets

Software Name	# bug reports	# product	# OS	# components
Apache Tomcat	1525	2	16	16
Eclipse	1674	2	17	13
Linux Kernel	1692	16	1	106

We first apply pattern matching to extract static meta information, as listed in Table 1. Then we process free text descriptions using tokenization and bag-of-words feature selection as described in section 3.2. The dimensionalities of the term feature space range from 4000 to 13,000 depending on the dataset. After the attribute data sources are combined, the final vector space to represent bug report instances includes static meta information and bag-of-words features.

4.3 Results and Analysis

Our experimental results show that BUGMINER is effective in automatic completion check, automatic redundancy check, and bug report trend analysis. The following subsections present the detailed results and analysis.

4.3.1 Classification for Missing Field Autocompletion

For missing field autocompletion, we train classification model on 80% of the data and do blind-test on the remaining 20% of the data. For example, for Apache Tomcat, we use 1220 (or 80%) bug reports as training data and use 305 (or 20%) bug reports as the testing data. Table 4 lists the classification results for the Tomcat version. The accuracy of the classification on testing instances is 99.02%. This means the automatic completion check engine can determine the product version highly accurately in this case.

Table 4. Classification results of products

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.991	0.014	0.996	0.991	0.993	0.989	tomcat 3
0.986	0.009	0.973	0.986	0.98	0.989	tomcat 7
0.99	0.012	0.99	0.99	0.99	0.989	Weighted Avg.

Table 5 lists the classification results for the operating system version for Tomcat. The accuracy of the classification on testing instances is 68.52%.

Table 5. Classification results of OS versions

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.888	0.449	0.758	0.888	0.818	0.719	all
0.356	0.081	0.432	0.356	0.39	0.637	linux
0.087	0.018	0.286	0.087	0.133	0.535	other
0.176	0.014	0.429	0.176	0.25	0.581	solaris
0.786	0.047	0.629	0.786	0.698	0.869	windows xp
0.685	0.294	0.632	0.685	0.647	0.696	Weighted Avg.

Table 6 lists the classification results for the software component related to the bug report. The accuracy of the classification on testing instances is 53.11%. The results show that it is relatively difficult to accurately determine the problematic component based on the bug reports in this case.

Table 6. Classification results of components

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.5	0.007	0.714	0.5	0.588	0.747	auth
0.868	0.067	0.73	0.868	0.793	0.9	catalina
0.2	0.039	0.313	0.2	0.244	0.58	config
0.368	0.037	0.583	0.368	0.452	0.665	connectors
0.5	0.003	0.5	0.5	0.5	0.748	encoding
0.622	0.041	0.676	0.622	0.648	0.79	jasper
0.667	0.003	0.667	0.667	0.667	0.832	manager
0.6	0.154	0.513	0.6	0.553	0.723	servlet
0.1	0.007	0.333	0.1	0.154	0.547	webapps
0.531	0.091	0.535	0.531	0.518	0.72	Weighted Avg.

We also did some experiments on the bug report datasets of Eclipse and Linux Kernel. Table 7 shows the summary of classification accuracy rates for the datasets tested. As the number of classes increases, the accuracy rate tends to decrease; nevertheless, the accuracy rates (*e.g.*, 53.11% for Tomcat's components) are relatively high if they are compared to the chance baseline (*i.e.*, probability is $1/n$ if there are n possible components).

4.3.2 Similarity Ranking

We first transform the historic training bug reports and the testing bug report to vectors using the vector space model.

Table 7. Summary of classification accuracy

Software Name	product	OS	components
Apache Tomcat	99.02%	68.52%	53.11%
Eclipse	97.90%	66.47%	67.37%
Linux Kernel	76.33%	N/A	58.88%

After the csv and kld value for each training bug report are calculated, all the training bug reports are then sorted in an descending order based on the csv value and in an ascending order based on the kld value. The bug reports at the top of the ranked lists are the most similar ones to the testing bug report.

Based on the heuristics from the experiments, we determine the value range parameters as $c\_r_1 = 0.2, c\_r_2 = 0.9, k\_r_1 = 2.0$, and $k\_r_2 = 10.0$ for Tomcat. Table 8 lists some sample results for a given bug report #393. From the results, the bug report #393 is highly likely to be a duplicate of some prior reports because there exists historic bug reports with $csv \geq 0.9$ and $kld \leq 2.0$ (*i.e.*, bug report #330 and #296). Furthermore, bug report #228 is likely to be a similar bug report of #393 because it has $0.7 < csv < 0.9$ or $2.0 < kld < 10.0$. To determine whether a new or given bug report is in fact a duplicate usually requires human judgment. Our manual verification shows that the similarity ranking results produced by BUGMINER are highly accurate despite the low quality data.

Table 8. Similarity ranking results

bug_id	csv	kld
330	0.928	1.940
296	0.917	0.816
228	0.717	9.868

4.3.3 Trend Analysis

We implement the bug report trend analysis based on the Weibull distribution. We first aggregate the historic data to compute a vector of the time (i-th week) and the number of bug reports whose first reporting date falls in the i-th week. Then a result vector returns the 95% confidence intervals for the estimates of the parameters of the Weibull distribution given the historic vector data. The two-element row vector estimates the Weibull parameter λ and k. The first row of the 2-by-2 matrix contains the lower bounds of the confidence intervals for the parameters, and the second row contains the upper bounds of the confidence intervals.

Table 9 shows the estimates of the Weibull parameters for Apache Tomcat 3. The value of k is less than 1, which indicates that the incidence rate decreases over time. The related curve fit is illustrated in Figure 2. The starting time, (*i.e.*, the 0 on the x-axis) is the week of August 25, 2000. The curve fit shows that the Weibull distribution closely resembles the actual bug report incidence distribution.

Table 9. Weibull parameter estimates

Software	λ	λ_{low}	λ_{high}	k	k_{low}	k_{high}
Tomcat 3	0.3885	0.2280	0.6621	0.2241	0.2041	0.2461

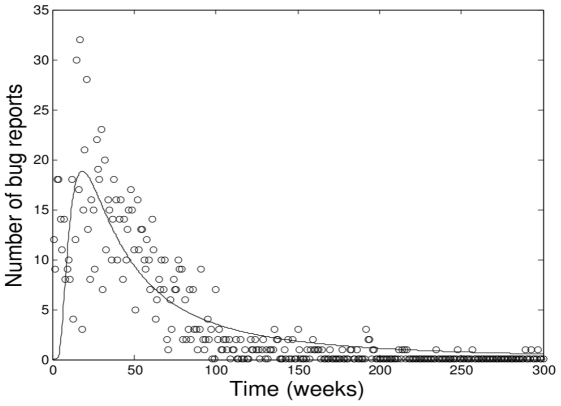

Figure 2. Weibull fit for Tomcat 3

5 Related Work

Some prior studies have been done on applying data mining on software engineering. [10] described the concept of software intelligence and the future of mining software engineering data. [19] presented a general overview of data mining for software engineering and described an example of duplicate bug detection using vector space-based similarity. [17] also described an approach to detect duplicate bug reports using both natural language and execution information. Our redundancy check engine uses both probability distribution-based KL divergence and vector space-based Cosine similarity ranking, instead of only vector space-based similarity. Furthermore, our approach provides a similarity ranking list that can be used for search, instead of only Yes and No on duplication check. [8] presented text mining of bug reports to identify security issues. Their work aims to identify security problems such as buffer overflow through mining the bug reports. Their purpose and techniques are different from our approach.

6 Conclusion

In this paper, we presented BUGMINER, a tool that is able to derive useful information from historic bug report database via data mining, use these information to do completion check and redundancy check on a new or given bug report, and to estimate the bug report trend using statistical analysis. We did empirical studies of the tool using several real-world bug report repositories. The experimental results show that BUGMINER is effective, easy to implement, and has relatively high accuracy despite low quality data. BUGMINER can be integrated into some existing bug tracking tools or software testing suites for more intelligent and cost-efficient software reliability analysis.

7 Acknowledgments

Wu and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 2 U54 CA121852-06.

References

[1] Apache Project. http://issues.apache.org/bugzilla/, 2011.

[2] Bugzilla. http://www.bugzilla.org, 2011.

[3] C. Cortes and V. Vapnik. Support-Vector Networks. *Machine Learning*, 20, Springer, 1995.

[4] T. M. Cover and J. A. Thomas. *Elements of Information Theory*, Wiley, 1991.

[5] Debian Bug Tracking System. http://www.debian.org/Bugs, 2011.

[6] Debian Project. Debian Bug report logs – #615000. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=615000, retrieved 2011-03-03.

[7] Eclipse. http://bugs.eclipse.org/bugs/, 2011.

[8] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, Cape Town, pp. 11–20, May 2010.

[9] A. Gilat. *MATLAB: An Introduction with Applications 2nd Edition*, John Wiley & Sons., July 2004.

[10] A. E. Hassan and T. Xie. Software Intelligence: Future of Mining Software Engineering Data. In *Proc. of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010)*, Santa Fe, NM, pp. 161–166, November 2010.

[11] J. Lafferty and C. Zhai. Document language models, query models, and risk minimization for information retrieval. In *Proc. of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 111–119, 2001.

[12] Linux Kernel. http://bugzilla.kernel.org/, 2011.

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

[14] S. E. Rigdon and A. P. Basu. Estimating the intensity function of a Weibull process at the current time: Failure truncated case. *Journal of Statistical Computation and Simulation (JSCS)*, vol. 30, pp. 17–38, 1988.

[15] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, v. 18 n. 11, pp. 613–620, November 1975.

[16] V. N. Vapnik. *The nature of statistical learning theory*, Springer-Verlag, New York, 1995.

[17] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*, pp. 461–470, ACM Press, 2008.

[18] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham. Weka: Practical Machine Learning Tools and Techniques with Java Implementations. In *Proc. of the ICONIP/ANZIIS/ANNES'99 Workshop on Emerging Knowledge Engineering and Connectionist-Based Information Systems*, pp. 192–196, 1999.

[19] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data Mining for Software Engineering. *Computer*, vol. 42, no. 8, pp. 55–62, IEEE Computer Society, Los Alamitos, CA, USA, 2009.