Learning from Bug-introducing Changes to Prevent Fault Prone Code

Lerina Aversano, Luigi Cerulo, Concettina Del Grosso
RCOST – Research Centre on Software Technology, University of Sannio
Via Traiano, 82100 Benevento, Italy
aversano@unisannio.it, lcerulo@unisannio.it, tina.delgrosso@unisannio.it

ABSTRACT
A version control system, such as CVS/SVN, can provide the history of software changes performed during the evolution of a software project. Among all the changes performed there are some which cause the introduction of bugs, often resolved later with other changes.

In this paper we use a technique to identify bug-introducing changes to train a model that can be used to predict if a new change may introduces or not a bug. We represent software changes as elements of a n-dimensional vector space of terms coordinates extracted from source code snapshots.

The evaluation of various learning algorithms on a set of open source projects looks very promising, in particular for KNN (K-Nearest Neighbor algorithm) where a significant tradeoff between precision and recall has been obtained.

Keywords
Software Evolution, Mining Software Repositories, Bug prediction

1. INTRODUCTION
Software faults affect negatively its quality assurance as the time and people available are usually not sufficient to eliminate all faults before a release. A technique that allows software engineers to identify the most fault-prone code is of great interest as it permits to concentrate the effort on such code, for example, with more test cases. On the other hand, a technique that prevents the introduction of fault-prone code offers another perspective of the same problem as it will inevitably reduce the presence of such fault-prone code. Moreover the concurrent action of a fault-prone code identification technique and a fault-prone code prevention technique should affect positively software quality in both testing and coding phases. In this paper we focus on predicting if a new change may introduce a bug, alerting developers to prevent the introduction of fault-prone code. Following a seminal work introduced in [14] we use a technique to identify bug-introducing changes to train a model and then predict whenever a new change is to be made if it introduces a potential bug or not. When developers make a change to a software system, either to add new functionality, restructure the code, or to fix an existing bug, they may inadvertently introduce a bug into the system. This is known as a *bug-introducing change*, the modification in which a bug was introduced into the software. Later, this bug may be discovered and tracked with a bug-tracking system and subsequently resolved by a developer with a commit into the versioning system. This is known as a *bug-fix change* which may includes, in the commit notes, the identifier of the bug report that was just fixed [9].

We exploit the information carried by source code changes which introduced a bug fixed later with another change. Such bug-introducing changes are important also for understanding properties of bugs and mining bug prone change patterns [17]. Our model uses a weighted terms vector representation of source code which has been applied with success in a number of contexts [2, 3], as it is able to synthesize with a strong mathematical structure the representation of source code. With such a representation a bug-introducing change can be computed with a vector difference and used as a feature vector of a learning algorithm. The basic intuition behind such approach is that weighted terms vector are able to capture change patterns and a learning model is able to classify those related to bug-introducing changes. This paper evaluate such intuition with various well known learning algorithms on two open source projects. Promising results have been obtained for KNN (K-Nearest Neighbor algorithm) with the best tradeoff between precision and recall (both at 58%) and Simple Logistic regression with the best precision (80%). Main contributions of this paper are: (i) a learning based bug prediction approach that can be a useful support to prevent the introduction of bug-prone changes;(ii) a quantitative evaluation of the proposed approach with respect to the adopted learning algorithms; and (iii) a qualitative evaluation about how bug-introducing causes can be discovered by inspecting into the learning model. The paper is organized as follows: the next Section gives an overview about the topics which the prediction model is based on; Section 3 introduces the prediction model and the learning algorithms we wish to evaluate; Section 4 evaluates the approach among different learning algorithms on two open-source projects; finally, related work and conclusions are given respectively in Sections 5 and 6.

2. BACKGROUND

In this Section we briefly introduce some topics which constitute the basis of the prediction approach presented in the following Section.

2.1 Snapshots extraction

A number of techniques have been presented in literature to extract from CVS/SVN repositories logical coupled changes performed by developers working on a bug fix or an enhancement feature [9]. Such techniques consider the evolution of a software system as a sequence of *Snapshots* $(S_0, S_1, \ldots, S_n)$ generated by a sequence of source code changes (Δ_1, Δ_2, $\ldots$, Δ_n), also known as *Change Sets*, representing the logical changes performed by a developer, for example, in terms of added, deleted, and modified source code lines. Approaches based on time–window consider a *Change Set* as sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 seconds [29]. Approaches based on time–warping consider a *Change Set* as the set of file revisions that change together almost all the time [6].

2.2 Finding Bug-introducing changes

A number of work in literature have been proposed to classify a source code change as a bug fix or not. There are methods that analyze log messages in two ways: searching for keywords such as 'Fixed' or 'Bug' [19] and searching for references to bug reports like '#12345' [9]. In addition to know when a bug is fixed, another question, introduced in literature, is to know when such a bug is introduced into the system. Kim *et al.* proposed a method to find the set of changes which introduce a bug fixed later [14]. We use such a technique to extract from the CVS/SVN repository, of a software system, the set of bug-introducing changes which provide the training set of a model to predict if a new change introduces a bug. Such a technique is briefly explained in Figure 1. Starting from a bug-fixing change, Δ_i, the set of preceding changes Δ_j, with $j < i$, are inspected to find those that contain the lines of code handled in the initial change, Δ_i. Such backward tracking is performed by using a line change detection algorithms which maps homologue lines of two subsequent snapshots [7].

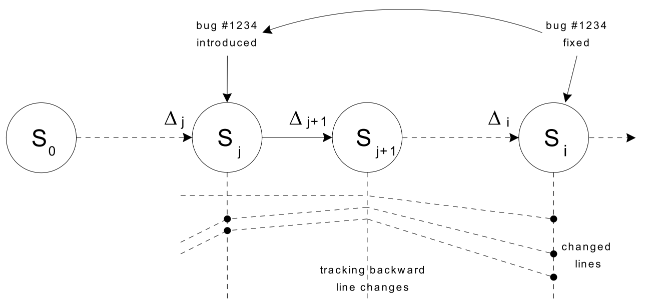


Figure 1: Identification of a bug-introducing change

As stated by their authors such approach exhibit a good precision but a limited recall as the causes of a bug may have hidden dependencies which are difficult to localize entirely.

2.3 Weighted terms vector representation of changes

A weighted terms vector is defined as set of coordinates in an n-vector space $(x_1, x_2, \ldots, x_n)$, where x_i is a real number representing the weight of the i^{th} term extracted from a source code document. We defined here a source code document as the unit we intend to represent, such as classes, files, packages, releases. Terms are extracted by considering a sequence of alphanumeric characters separated by non-alphanumeric characters including for example, variables names and types, language keywords, function/method names, comments. From such extraction process code comments are discarded as well as blank lines and large string literals. The weight x_i is defined by the product: $tf_i \cdot log(idf_i)$, where tf_i (i.e., *term frequency*) is the number of term occurrence divided by the total number of terms in the document, and idf_i (i.e., *inverse document frequency*) is the total number of documents divided by the number of documents such term appears.

Such a vector based representation permits to consider a software snapshot as a weighted terms vector in an n-dimensional space, where n is the total number of terms extracted from the overall set of snapshots (i.e. the vocabulary). Vector algebra allows to compute the representation of a source code change, Δ_j, as the vector difference between S_j and S_{j-1} (see Figure 2). Coordinates of such a vector difference are real numbers which can be positive, negative, and zero, when such code element coordinates has been added, deleted, and left unchanged respectively.

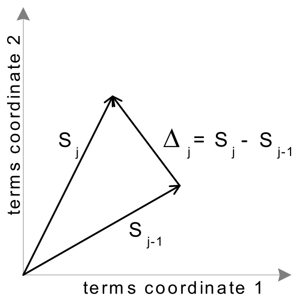


Figure 2: Representation of a change: vector difference between two subsequent snapshots

Vectors representing software changes have some interesting properties we briefly list in the following:

- They are insensible to large used code snippets due to the *idf* factor, for example composed of very frequent language keywords (ex. java *import* keyword) and/or domain specific code (ex. logging) occurrences;
- They are able to capture project-specific change patterns, such as particular co-occurrences of API usage patterns and pre/post conditions;
- They are independent from the programming language as they does not consider the syntactic information of source code;

3. PREDICTION MODEL

A prediction model is a mathematical model that makes predictions based on correlations or relationships among features [26]. After the prediction model is built, one sample at a time is loaded into the prediction model and processed to make a prediction of that sample. In the context of a software development process this means that when a change is to be performed by a developer a check point can be arisen, which evaluate if such a change may introduces a bug or not. Obviously the model cannot tell us which are the motivations of a bug-introducing change but it can alert developers that such a change he/she is currently typing could be risky or not. If it is the case of potential bug-introduction the developer can perform a more accurate review of s/he's code before committing the change into the system.

Such kind of assistance mechanism, which is a challenging issue in the future of programming environments [28], can be applied in particular during the coding phase and we belief that it could be crucial to reduce the effort of bug fixing performed usually during the testing phase as many of such bugs may be detected before. The evaluation of such effort reduction, such as the reduction of new bugs discovered in the case of using the prediction model and not, is a future work. In this paper we focus on evaluating the prediction capability of such a built model.

The prediction model can be synthesized as follows:

$$y = SLA(\Delta_{new}, \Lambda)$$

where, Δ_{new} is the incoming new change, Λ is the set of past performed changes (i.e. learning set) for which is known if introduces a bug or not, SLA is a supervised learning algorithm chose from the literature of machine learning algorithms [26], and y={BUG, ¬BUG} is the dichotomic dependent variable with two possible outcomes: introduces or not a bug. A supervised learning algorithm generates a function that maps inputs to desired outputs, which in general is a classification problem. The learner is required to approximate the behavior of a function which maps a vector into one of several classes by looking at several input-output examples of the function. In our case the number of classification classes are two: BUG and ¬BUG.

In the following a brief description of the learning algorithms we evaluated in the case study. We selected the algorithms which are more suitable for a categorical classification problem and which have reported a valuable performance. More information can be found in their respective references.

3.1 K-Nearest Neighbor

K-Nearest Neighbors (KNN) classifier is a type of instance-based learning for classifying objects based on closest training examples in the feature space [1]. The training examples are mapped into multidimensional feature space which is partitioned into regions by class labels of the training samples. A point in the space is assigned to the class if it is the most frequent class label among the k nearest training samples. Usually a weighted Euclidean distance is used to compute the closeness to samples. The number of neighbors, k, can be set both as a parameter or can be selected considering the mean squared error for a given training set.

3.2 Simple logistic regression

Logistic regression models are particulary suitable in the case of dichotomous dependent variables [16]. In the case of Simple logistic the model is fit with a simple regression function, usually a sigmoidal, used as a base learner. The optimal number of iterations can be detected automatically and can be set in order to minimizes the root mean squared error on the training set.

3.3 Multi-boosting

Multi-boosting is an improvement of the adaptive boosting (ADABoosting) meta-algorithm which has subordinate algorithms as variable and replaceable parameters [25]. The basis of such algorithms are the boosting stages, which increments iteratively the performance of the current learned function. At every stage a weak learner is trained with the data and then its output is added to the learned function, with some strength proportional to how accurate the weak learner is. Multi-boosting uses the C4.5 as the base learning algorithm with a wagging mechanism which provide a fast convergence to the optimal learner function.

3.4 C4.5

C4.5 is a decision tree generating algorithm, based on the ID3 algorithm [20]. A decision tree is basically a tree in which each branch node represents a choice between a number of alternatives, and each leaf node represents a decision. They has been successfully used in expert systems in capturing knowledge. The main task performed by a decision tree generating algorithm, such as C4.5, is using inductive methods to the given values of attributes of an unknown object to determine appropriate classification according to decision tree rules. In particular C4.5 allows to avoid overfitting the data, to determine how deeply to grow a decision tree, to reduce error pruning, rule post-pruning, to choose an appropriate attribute selection measure, to handle training data with missing attribute values, and to handle attributes with differing costs.

3.5 SVM

Support Vector Machine (SVM) is a kernel based learning algorithm that use the idea of structural risk minimization from computational learning theory [8]. The classifications main purpose is to obtain a good generalization performance, that is a minimal error classifying unseen test samples. Basically SVM map input vectors to a higher dimensional space where a maximal separating hyperplane is constructed. Two parallel hyperplanes are constructed on each side of the hyperplane that separates the data. The separating hyperplane is the hyperplane that maximizes the distance between the two parallel hyperplanes. Given a labeled set of training samples a SVM classifier finds the optimal hyperplane that correctly separates (classifies) the largest fraction of data points while maximizing the distance of either class from the hyperplane (the margin). An assumption is made that the larger the margin or distance between these parallel hyperplanes the better the generalization error of the classifier will be.

Table 1: Case study history characteristics

System	Snapshots/ Changes	Releases	KNLOC	Classes
JHotDraw	132	5.2–5.4B2	13.5–36.3	164–489
DNSJava	1204	0.3–2.0.2	5.5–25.3	55–179

Table 2: Data-set characteristics of each system

	# of changes		
System	bug-fix	bug	¬bug
JHotDraw	38	19 (14%)	113 (86%)
DNSJava	114	263 (22%)	941 (78%)

4. CASE STUDY

We selected two open-source systems, JHotDraw and DNSJava which can be classified as small size systems. We extracted from such systems only the HEAD development trunk (i.e., excluding branches) by using the time-window heuristic [9] and ignored changes that involve more than 30 source code files in order to exclude large CVS maintenance activities. Table 1 reports for each system, the number of extracted snapshots, the range of analyzed releases, the minimum and maximum number of non commented lines of code (KNLOC), and the minimum and maximum number of classes (excluding anonymous-classes).

JHotDraw[1] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of showing the Design Pattern Programming in a real context. We extracted a total of 132 snapshots from release 5.2 to release 5.4 BETA2, in the time interval between March 2001 and February 2004. In that interval the size of the system grew almost linearly from 13.5 KNLOC at release 5.2 to 36.5 KNLOC at release 5.4 BETA2.

DNSJava[2] is an open-source Domain Name Server written in Java. The project started in September 1998. It comprises classes for handling DNS names, records, addresses, and for caching name resolutions. The interval of observation considered ranges from March 1999 (release 0.3) to June 2006 (release 2.0.2). The total number of releases produced in this range is 52 including pre and beta releases. The CVS system is managed by a single user account, probably because the submission of changes are performed in a strictly controlled way. The number of classes grew from 55 to 179 in a non linear fashion, while the number of non commented lines of code ranges between 5000 and 25000. The total number of snapshots extracted from the HEAD development trunk is 1204.

We applied the prediction model introduced in Section 3 on both JHotDraw and DNSJava data-sets and evaluate the performance by using different learning algorithms. Table 2 summarizes such data-sets characteristics showing the number of changes which fix a bug, introduce a bug (bug), and those that don't introduce bugs (¬bug). Beside the good precision of the bug-introduction detection approach, we inspected manually such data-sets in order to eliminate all false positives and discarded those change commits which include more than 30 different source files which may introduce noise in the change vector representation. False negatives have been ignored as they are difficult to retrieve manually. In any case their absence should not strongly affect the results we have obtained. Figures 3(a) and 3(b) show the density distribution among snapshots of bug-introducing changes. DNSJava exhibit an evident decreasing trend, while JHotDraw have a peak in the middle stage of development and then still decreases. This does not mean that the most recent changes does not introduce bugs but, as the technique briefly introduced in Section 2.2 is able to identify bug-introducing changes by starting from bug-fixed changes, more recent bug-introducing changes could be identified only later, when new bugs are discovered and fixed.

We used *Weka*[3] which provides a collection of machine learning algorithms for data mining tasks [26] and a number of model validation options. We chose to perform two model validation options discussed in the next two subsections. The first consists in a 10-fold cross-validation, while the second consists in an incremental learning validation which is more adherent to a real development context. Performances has been calculated with the precision and recall for each of the two possible outcome values (bug and ¬bug). This is because the distribution of bug-introducing and not-but-introducing changes are not equal, and not-bug-introducing changes is usually more higher than bug-introducing changes (see Table 2). Precision and recall are defined respectively as:

$$P = \frac{CP}{CP + FP} \qquad R = \frac{CP}{CP + FN}$$

where CP is the number of correct positives, FP is the number of false positives, and FN is the number of false negatives.

4.1 Cross-validation

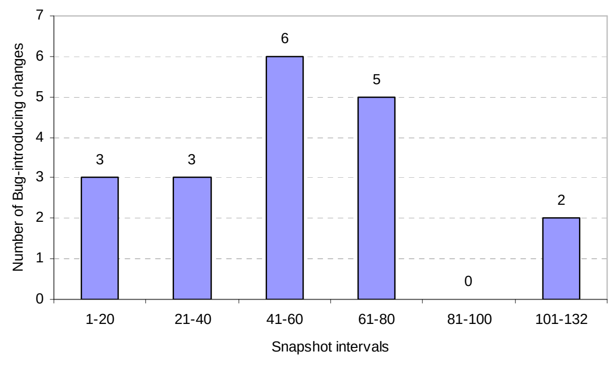
In order to compare performances of the different methods, a 10-fold cross-validation [24] for each learning algorithm was performed with the dataset of both JHotDraw and DNSJava. Percentages of correct classification for each possible outcome are reported in Tables 4 and 5. The precision and recall have been used as performance measures. In particular, considering that the main objective of using such prediction model is to correctly predict if a change introduces a bug, the main focus is on the Bug outcome.

The settings options of each algorithm have been fixed after numerous trials in which we used the real data set. Settings which have revealed the best performance are the same in both system. Table 3 reports a brief description of the setting used for each prediction model.

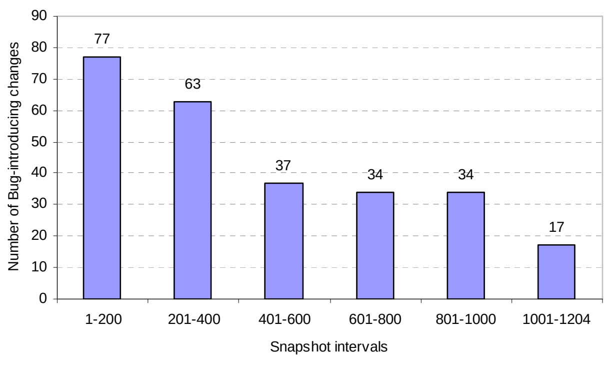
We experienced that clearly emerge a best algorithm, which is KNN, performing significantly better than others with a good tradeoff between Precision and Recall, both for bug and ¬bug classifications. Moreover this result is confirmed in both JHotDraw and DNSJava. The increment with respect to other learning algorithms is of about 10%. The worst performance is obtained with SVM probably because

[1] *http://www.jhotdraw.org*
[2] *http://www.dnsjava.org*

[3] http://www.cs.waikato.ac.nz/ml/weka/



(a) JHotDraw



(b) DNSJava

Figure 3: Distribution of bug-introducing changes among snapshots

Table 3: Prediction model settings

Prediction Model	Settings
KNN	A number of K=10 neighbor has been used with a mean squared error validation on the current training set selecting the best one.
Simple Logistic	The number of iterations is chosen that minimizes the root mean squared error on the training set.
Multi-boosting	A decision tree algorithm has been used as the base multi-boosting classifier, the number of maximum iterations performed has been 10, and for weighing pruning has been used a threshold equals to 100.
C4.5	The confidence factor used for pruning has been fixed to 0.25, the minimum number of instances per leaf has been fixed in 2, one fold has been used for pruning and 2 for growing the tree, and when pruning a subtree raising operation has been considered.
SVM	The constant rate and percentage rate at which attributes are eliminated per invocation of the support vector machine have not been changed from default value (1 and 0 respectively). The epsilon value for round-off error has been set to 1.0e-25 while the tolerance parameter to 1.0e-10, both are default values.

to the huge number of feature vector attributes, which are 1846 and 2856 respectively for JHotDraw and DNSJava. An average performance is obtained for both C4.5 and Multi-boosting which exhibit similar precision/recall values. Simple logistic exhibits the best precision but almost the worst recall for both systems. We noted that both recall and precision of Simple logistic is highly correlated with a size metric base predictor, such as the number of source code lines involved in a change, as an indicator of bug-introducing changes. This is probably related to the implicit model construction of a Simple logistic function. Usually size metrics is a raw indicator (good in some cases) of fault prone code [4], i.e. large changes have a higher probability to introduce

Table 4: Model performances on JHotDraw

Prediction Model	bug		¬bug	
	Precision	Recall	Precision	Recall
KNN	58.8%	58.8%	93.3%	93.3%
Simple Logistic	80.0%	21.1%	88.2%	99.1%
Multi-boosting	67.7%	21.2%	88.8%	98.8%
C4.5	50.0%	13.0%	89.9%	98.7%
SVM	50.0%	10.5%	86.2%	98.1%

Table 5: Model performances on DNSJava

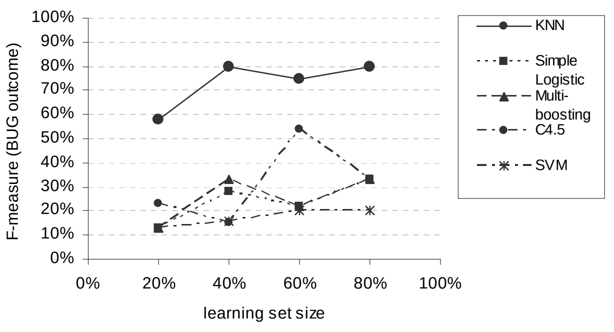
Prediction Model	bug		¬bug	
	Precision	Recall	Precision	Recall
KNN	69.4%	23.1%	87.1%	98.1%
Simple Logistic	60.0%	13.7%	80.2%	97.4%
Multi-boosting	52.0%	30.0%	82.2%	92.2%
C4.5	48.8%	29.0%	82.0%	90.0%
SVM	39.9%	31.6%	81.9%	86.7%

a bug than a smaller one, but in this case they exhibit a very low recall. Such behavior let us to consider that the vector representations used for changes are able to capture key information, ignored by size metrics, which contributes positively to identify bug prone changes.

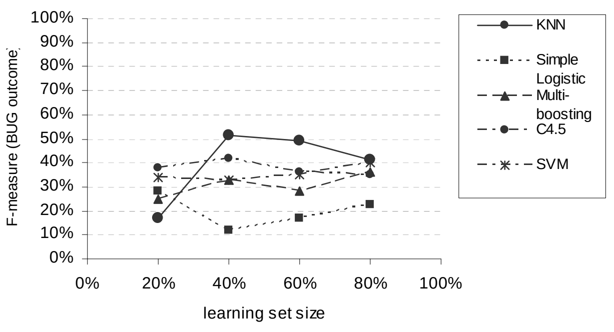
Overall, the results achieved are encouraging. More focused investigation are required (for example by involving developers in a real software project) but the measure obtained for precision and recall is an initial attempt to confirm the proposal to support the prediction of introducing bugs during the coding development phase with the purpose of preventing fault prone code generation.

4.2 Incremental learning validation

Incremental learning validation tries to simulate a real software change process: a new change is classified according to the changes performed previously and considering all of them as the training set. To perform such validation we consider snapshots ordered by the commit time-stamps split, in training and test sets, at regular intervals. Results, for a learning set that grows chronologically, are shown in Figures 4(a) and 4(b) which report, on the y-axis, the F-measure



(a) JHotDraw



(b) DNSJava

Figure 4: Incremental learning performance

for the outcome that a bug is introduced (y=BUG). The F-measure is the balanced harmonic mean of Precision (P) and Recall (R) [21]. It is given by the following expression:

$$F = \frac{2 \cdot R \cdot P}{P + R}$$

Both JHotDraw and DNSJava exhibit a transient oscillation at the beginning which tend, almost asymptotically, to the performance values obtained with a 10-fold cross-validation reported in Section 4.1. This is coherent with the development process as more bug-introducing change examples contribute a better prediction performance still around the value obtained with a 10-cross validation. Some exceptions of such trend can be observed but we ascribe such behavior mainly to the dataset distribution characteristics where the number of bug-introducing changes almost decreases as development proceeds (see Figures 3(a) and 3(b)). Moreover for JHotDraw the number of samples for the training set dataset may not contain a valuable number of examples.

4.3 Threats to validity

This section discusses threats to validity that can affect the results reported in this Section, following a well-known template for case studies [27].

Systems examined might not be representative (external validity). We considered two different software systems, differing for their domain (graphical editor vs. dns server) but having almost the same size and both are open source projects. We obtained some common findings and some results peculiar to each system. Nevertheless, it would be desirable to analyze further systems — also developed in different programming languages and with different size — to draw more general conclusions.

Bug fixing changes are incomplete (external validity). Only a sub set of all faults can be extracted from a project, which is typically 40%-60%, even though with a high quality of historic data. However, we are confident that the model performance improves with the quality of the dataset.

Change sets and bug-introducing changes identification might not be accurate (construct validity). The identification of change sets, i.e. logical coupled changes performed by developers during their maintenance activity, and bug-introducing changes, described in Section 2, might not be accurate as they are based on heuristics which in some cases could be not valid. However in both cases such events are very rare in the project we have considered.

Computational complexity (construct validity). A limitation of learning algorithms is that the training phase may be computational expensive and may not be used in a real environment. However, this is a partial limitation as the training phase can be performed while development proceeds.

Regarding *reliability validity*, the source code of the two systems is publicly available and the way our analyses were performed is described in detail in Section 3.

Threats to *internal validity* did not affect this particular kind of study. Error measure, such as those deriving, for example, from the heuristics used to classify bug-fixing changes has been reduced considerably with a partial inspection of such changes.

5. RELATED WORK

The problem of fault prediction has been mainly addressed by identifying software entities, such as modules, which should manifest problems. There are methods based on software quality metrics and those based on change history analysis.

Gyimóthy *et al.* performed an empirical validation of how object-oriented metrics (Chidamber and Kemerer) are correlated with software faults [11]. Khoshgoftaar and Allen proposed a quantitative software quality model to predict the rank-order of modules according to a quality factor, such as the number of faults [12]. Bell developed a negative binomial regression model to predict the expected number of faults in each file of the next release of a system on the basis of code attributes extracted from previous releases [5].

Graves *et al.* assumed that modules that were changed recently are more fault-prone than modules that were changed a long time ago [10]. Livshits and Zimmermann proposed DynaMine, a tool that analyzes source code check-ins to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns [17]. Śliwerski *et al.* show how to automatically locate fix-inducing changes and found that those performed on Friday induce more bug fixing [23]. Kim *et al.* present a bug finding algorithm using the history of bug fixes [13]. The results demonstrate, congruently with our findings, that bug fix patterns occur frequently enough to be useful as a bug detection technique. The main difference with our approach is in the representation of bug-introducing changes, in our case, n-dimensional term vectors, more suitable to be used as feature vectors in a learning algorithm. Śliwerski *et al.* developed a prototype, HATARI, to detect those locations where changes have been risky in the past and makes this risk visible for developers by annotating source code with color bars [22]. Kim *et al.* analyzed the version history of 7 software systems to predict the most fault prone entities and files with a cache based approach [15]. In particular they showed, by consulting the cache at the moment a fault is fixed, how a developer can detect likely fault-prone locations. The difference with our approach is that they assume the locality (spatial and temporal) of a bug location, while we consider the representation in an n-dimensional space of fault prone location inside a co-change (which is near in time and may be spread in space). Mizuno *et al.* presented a related approach which use a spam filtering detection algorithm on source code text to classify fault prone software modules [18]. With respect to our approach they consider an entire software modules source code text for training, instead, we concentrate on text belonging to a software change which could be spread also on different modules.

6. CONCLUSIONS

If we can prevent the introduction of faults into a system then we are able to reduce the number of bug prone modules and consequently the density of bugs. The quantitative evaluation of such reduction is object of future work. In this paper we introduced a model to represent software changes as elements of an n-dimensional space which can be used as feature vectors for training a classifier. We feel that such a representation encapsulate change patterns effectively as it is independent from the programming language used and subsumes to a strong mathematical structure. We tested the prediction model on two open source systems, JHotDraw and DNSJava, considering different learning algorithms. Results feels promising, in particular for KNN (K-Nearest Neighbor algorithm) where a significant tradeoff between precision and recall has been obtained.

Beside to perform a more accurate evaluation with more machine learning algorithms and case studies, we see room of improvements in the following main topics.

Representation of changes. In this paper we consider a feature vector which attributes are terms extracted from source code. We intend to move along two main direction of improvement. First, a more accurate representation, with a more accurate parsing, including also syntactic properties of source code rather than only language keywords and identificators, could be useful to model also syntactic fault prone changes such as assignment errors and so on. Second, other vector based representation may be interesting to evaluate, such as vectors composed of source code metrics attributes.

Fault prone code. Detecting fault prone modules is a "diagnosis/cure like" approach which leads the attention of developers to concentrate on such modules with more test cases, the "cure of the illness". Instead, the approach presented in this paper focus on changes which may introduce a bug into the system preventing, as a "preventive like" approach, the generation of fault prone code. We intent to exploit both point of views and in particular how bug-introducing changes can help in identify fault prone code.

Indirect bug-introducing dependencies. The approach used to detect bug-introducing changes considers those strongly related with bug-fixing changes by inspecting the change history of their lines of code. Such approach assumes a strict (historical change) relationship between the cause of a bug and its fix. In general this is not true as there could be conditions such that code introduced somewhere may causes bugs elsewhere. To consider such indirect bug-introducing dependencies historical changes should be related with source code dependencies such as those originated from software slicing.

Comparison with source code metrics. Source code metrics, such as those related to size and complexity, are widely used to predict fault prone code. We wish to perform a comparison with such metrics in order to put in evidence which are their respective advantages and drawbacks.

7. ACKNOWLEDGMENTS

The work described in this paper is supported by the project METAMORPHOS (MEthods and Tools for migrAting software systeMs towards web and service Oriented aRchitectures: exPerimental evaluation, usability, and tecHnOlogy tranSfer), funded by MiUR (Ministero dell'Universitá e della Ricerca) under grant PRIN2006-2006098097.

8. REFERENCES

[1] D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, 1991.

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

[3] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE '04: Proceedings of the International Workshop on Principles of Software Evolution*. IEEE Computer Society, 2004.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[5] R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[6] S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[7] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the 29th International Conference on Software Engineering Workshops*. IEEE Computer Society.

[8] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[9] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society, 2003.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

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

[12] T. M. Khoshgoftaar and E. B. Allen. Ordering fault-prone software modules. *Software Quality Control*, 11(1):19–37, 2003.

[13] S. Kim, K. Pan, and J. E. E. James Whitehead. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45. ACM Press, 2006.

[14] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[15] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, 2007 (to appear). IEEE Computer Society.

[16] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. *Mach. Learn.*, 59(1-2):161–205, 2005.

[17] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[18] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *MSR '07: Proceedings of the 29th International Conference on Software Engineering Workshops*. IEEE Computer Society.

[19] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[20] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[21] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval*. Addison Wesley, 1999.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: raising risk awareness. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 107–110. ACM Press, 2005.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

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

[25] G. I. Webb. Multiboosting: A technique for combining boosting and wagging. *Mach. Learn.*, 40(2):159–196, 2000.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.

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

[28] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In L. Briand and A. Wolf, editors, *Future of Software Engineering*. IEEE Computer Society, 2007.

[29] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.