

---

# Learn to Predict “Affecting Changes” in Software Engineering

---

Xiaoxia Ren

XREN@PAUL.RUTGERS.EDU

Department of Computer and Information Sciences, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854

## Abstract

This paper explores a new possible application of machine learning algorithms: change impact analysis. Two versions of a java project were analyzed and their differences were decomposed into a unique set of atomic changes. The execution behavior of a test may have been affected by the applied changes. We call those atomic changes that were responsible for the test’s modified behavior “affecting changes”. Our data were collected from M. Ernst’s Daikon system of 2002. Five different classifiers were performed on the data, and we use different measurements to evaluate different classifiers. The data had some specific characteristics: 1. on average, about 5.9% of changes are affecting changes for a specific test; 2. the changes in two time intervals seldom overlap. Due to these two reasons, there is not a satisfying classifier to help us predict the affecting changes based on the history data.

## 1. Introduction

Software systems evolve over time in order to adapt to environment changes and to add desired functionality. Graceful software evolution requires that only expected changes in functionality occur when the programmers make changes to the source code. However, in object-oriented programs, the extensive use of subtyping and dynamic dispatch make program understanding a non-trivial task. Adding a new method in an existing class may affect the dispatch of virtual method calls throughout the whole program. As a result, program maintenance becomes a headache-inducing job because of the unforeseen impacts of software changes might have. Change impact analysis is intended to help programmers to determine the effects of a set of source code changes and thus enable safe code enhancement.

*Chianti* is a prototype change impact analysis tool (Ren and Shah 2003). Here, I only give a short introduction of this impact analysis, whose results were used as the experiment data in this paper. First, two versions of a java project were analyzed and the differences were decomposed into a unique set of interdependent atomic changes  $A$ . Second, for a given set  $T$  of (unit or

regression) tests, the analysis determines a subset  $T'$  of  $T$  whose execution behavior may have been *potentially affected* by the changes in  $A$ . We got such information by correlating the changes in  $A$  against the call graphs for the tests in  $T$ . Third, for a given test  $t_i \in T'$ , programmers may want to know which of the atomic changes cause the functionality change of this test. The analysis finds a subset  $A_i$  of  $A$  that contains all the changes that affect  $t_i$  by correlating the call graph for  $t_i$  with the changes in  $A$ . We call these edits “affecting changes”.

Given the data collected from *Chianti*, our goal is to learn whether it is possible to predict the affecting changes for a specified test based on the updating history. In this paper, five different classifiers applied to the data and several measurements were used to evaluate the results.

Section 2 describes the data format collected from *Chianti*. In Section 3, the experimental setup is presented. Section 4 evaluates different classifiers using different measurements and gives some thoughts on applying machine learning algorithms in software engineering. Related work is summarized in Sections 5.

## 2. Data Collection from *Chianti*

*Chianti* analyzed two versions of a java project and decomposed the differences into a unique set of atomic changes. Given a specific test, each change can be classified into one of two classes: “yes” class and “no” class, by checking whether an atomic change is an affecting change of this test. Detailed algorithms were described in Ren and Shah (2003).

The atomic change is represented as a tuple  $\langle category, packageName, className, changeSignature \rangle$ . *Category* specifies the kind of this atomic change. For example, it could be “adding a class”, “changing a method”, “deleting a field”, etc. The other three elements are self-explanatory: *packageName* and *className* specify the package name and class/interface name where this atomic change occurs; *changeSignature* is some kind of signature of the atomic change. For example, the *changeSignature* of an atomic change related to a class is the name of this class; the *changeSignature* of an atomic change related to a method is the signature of this method in java binary code, and so on.

```

@relation DifferTester.testEmptyPpt1.affectingChanges
@attribute category { ... }
@attribute packageName {...}
@attribute className {...}
@attribute signature { ... }
@attribute isAffecting { yes, no }
@data
changeMethod,daikon,PptSlice,check_modbits(),no
addClass,daikon,Dataflow,$1,no
addClass,daikon.diff,none,DepthFirstVisitor,yes
addField,daikon.diff,Diff,PPT_COMPARATOR(java.util.Comparator),yes
deleteField,daikon,PptSlice,parent(daikon.Ppt),no

```

Figure 1. An example of data collected from Chianti

Figure 1 gives an example of an arff file collected from *Chianti*. For simplification, some attributes values were not listed. This relation is called “DifferTester.testEmptyPpts1.affectingChanges”. The first instance means that changes made to the method *check\_modbits()* defined in class *daikon.PptSlice* does not affect the behavior of the test *DifferTester.testEmptyPpts1*, while the new added class *daikon.diff.DepthFirstVisitor* does affect the function of this test.

### 3. Experimental Setup

The experimental data collected on versions of the Daikon system by Ernst et al. (2000), extracted from the developers’ CVS repository. The Daikon CVS repository does not use version tags, so I partitioned the version history arbitrarily at week boundaries. All modifications checked in between one week boundary and the next were considered within one edit cycle whose impact was to be determined. However, in cases where no editing activity took place in a given week, I extended the interval with one more week, until it included changes. The data collected covers all the updates in year 2002, during which there were 39 intervals with editing activity.

In Daikon system, there was a collection of unit tests associated with each version extracted. The experimental results reported in the next section were performed on test “daikon.test.diff.DiffTester.testEmptyPpts1.affectingChanges”. In total, there are 19,742 atomic changes (instances), and for this specified test 1,165 of them are affecting changes (“yes” class), which means about 5.9% of the atomic changes potentially change the function of this test.

I gathered 39 arff files for each test from 39 intervals with editing activity. The number of instances of these files is

diverse, from 3 to more than 4,000. To make the data more suitable for learning, I combined the instances in a month together and if there was not enough editing activity (fewer than 800 atomic changes) in a given month, I extended the interval by months until it included enough changes. This procedure produced 8 arff files for the experiments. Depending on the source of training data, the experiments could be performed in two ways: 1. Training the classifier using data from one single month, then using the next month’s data for testing. 2. Training the classifier using all data from January to the current month, then testing this model on the next month’s data. On average, the second way of training provided a better result.

## 4. Evaluation

In this section, I will compare and evaluate the experimental results of the cumulative training, that is, using the data up to current month as the training data and then applying the training model to the test data. All the machine learning algorithms used in this paper are provided in Weka, a java system developed at Univ. of Waikato in New Zealand. The five classifiers I used for comparison are: ZeroR, J48, IB1, IBk and NaiveBayes. ZeroR is a very primitive scheme, which simply predicts the majority class in the training data. Since there are only 5.9% of the instances are in “yes” class, ZeroR always predicts “no”. J48 implements an improved C4.5 algorithm. In the experiment, I use reduced-error pruning to optimize performance. IB1 and IBk implement 1-nearest-neighbor and k-nearest-neighbors classifier respectively. In the experiment, k is initialized to 5 and the algorithm itself used cross-validation to determine the best value of  $k^*$  between 1 and k. The distance weighting was set to  $1/\text{distance}$ . NaiveBayes implements the probabilistic naïve Bayesian classifier.

### 4.1 Experimental Results

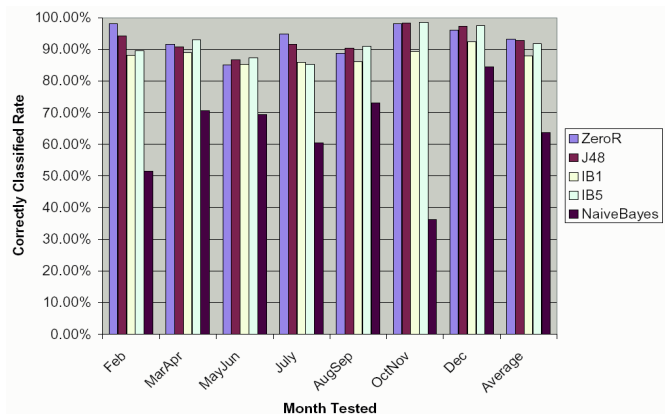


Figure 2. Comparison of 5 Classifiers with respect to accuracy

Figure 2 shows the comparison of 5 classifiers by accuracy. We found that on average, the most primitive

scheme ZeroR gets the best prediction rate (93.2%). J48 and IB5 also get very similar results (92.8% and 91.7% respectively), while the scheme NaïveBayes only gets 63.7% of the predictions correct.

Our goal is not only high accuracy, but also high precision of predicting “affecting changes” (“yes” class). So, we need an other measurement to evaluate these classifiers. Two possible measurements are True Positive (TP) rate and Precision of “yes” class. TP is also called “recall” in some contexts.

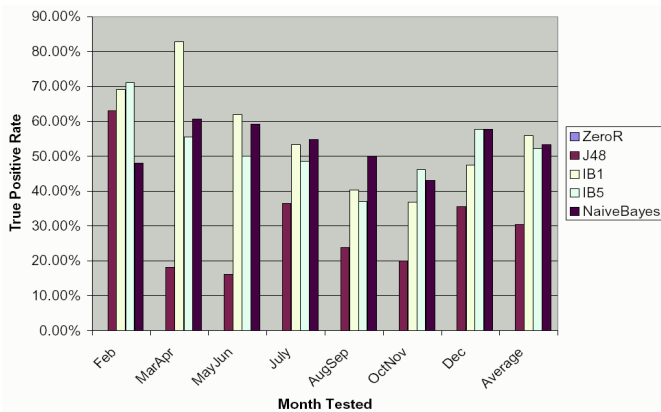


Figure 3 Comparison of 5 Classifiers with respect to True Positive (TP) rate for “yes” class.

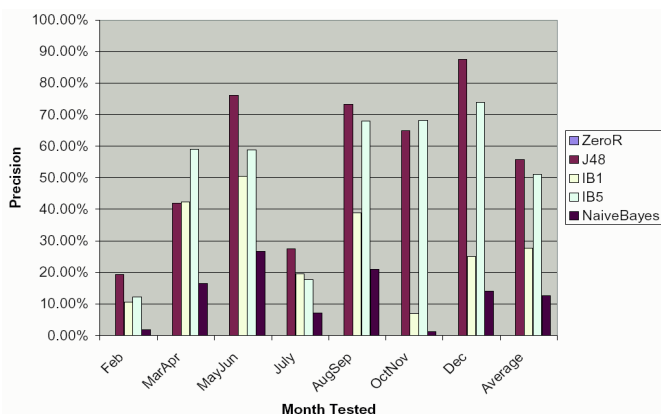


Figure 4 Comparison of 5 Classifiers with respect to Precision for “yes” class.

TP rate for class “yes” is the proportion of examples that were classified as class “yes” among all examples that truly have class “yes”. Precision is the proportion of the examples that truly have class “yes” among all those that were classified as class “yes”. In other words, the TP rate tells us the fraction of the “yes” instances that are captured, while precision gives information about how many of the predicted “yes” instances are really “yes” instances. For example, if in the test data, there are 100 instances really in the “yes” class. One scheme applies the training model on this data set and classified 130 of them as “yes”. If 90 of these 130 are correctly classified, then TP rate is  $90/100 = 0.9$  and the precision is  $90/130 = 0.69$ .

A good classifier should get both higher TP rate as well as higher precision.

Figure 3 and Figure 4 give the TP rate and precision comparison for the “yes” class of these five classifiers. Since ZeroR scheme always predicts instances as “no” class, the TP rate and precision are 0 and it is not visible in these two figures. From Figure 3, we notice that IB1, IBk and NaiveBayes all achieve more than 50% TP rate, which means they could capture more than half of the “yes” instances in the test data, while J48 could only capture 30.5% of the “yes” instances. However, when considering the precision shown in Figure 4, we found that J48 and IBk provide more than 50% precision, but IB1 and NaiveBayes schemes perform poorly.

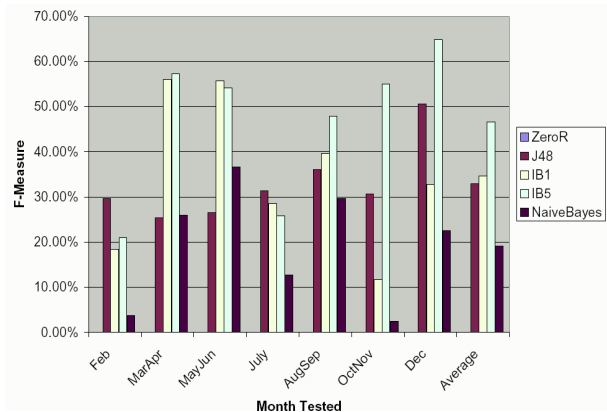


Figure 5 Comparison of 5 Classifiers with respect to F-Measure for “yes” class.

A good scheme should capture many true positive instances while at the same time, should misclassify as few instances as possible. The F-measure is such a kind of combined measure of TP rate and precision, it is get from equation  $2 * Precision * TP / (Precision + TP)$ . Figure 5 gives the F-measure for “yes” classes of five classifiers. On average, IBk performs better than other classifiers.

## 4.2 Result Analysis and Some Thoughts

The experiments show that, on average, the best TP rate or precision of “yes” class we get on the test data is less than 56%. Using the F-measure to evaluate the results, the number is worse, about 47%. There are two reasons that bring about this unsatisfying result.

First, change activity on Daikon system was quite diverse. Daikon was actively being developed in 2002, and increased in size from 48K to 121K lines of code. More significant are the program-based measures of growth, from 357 to 755 classes, 2409 to 6284 methods, and 937 to 2885 fields. There is not much overlap of change activities in different months, so the similarity between training data set and testing data set is small. For example, a bunch of instances are related to adding activity and

each “adding” instance could only appear once in the data set and won’t be repeated thereafter.

The other reason is that the percentage of “yes” class is too small, about 5.9%. Furthermore, 32% of the “yes” class was related to adding activity, which occurs only once. So, it’s hard for classifiers to build graceful rules for prediction.

Given the experiments performed and the data collected, we could draw a conclusion that for an actively developed system, machine learning algorithms may not be appropriate for change impact analysis, especially in predicting the affecting changes. However, if the system was already in a relatively stable stage, machine-learning algorithms may help us in predicting affecting changes, and thus helping in building some useful debugging tools.

Although it doesn’t perform well in predicting affecting changes, I found an other learning scheme, association rules, works well in helping to search for underlying rules. For example, when applying “Apriori” associator in Weka on the whole dataset, rule “packageName=jtb 1569 ==> isAffected=no 1569 conf:(1)” is mined. Such kinds of rules could be used to filter out some “no” instances from the data set, and maybe it helps us get a better results. More important, a rule like this gives programmers a hint of the coverage of this specified test, which is a significant research branch of software engineering.

## 5. Related Work

In recent years, there are trends toward fusion of machine learning and software engineering. Machine learning algorithms have been used in a variety of software engineering applications and impressive results have been reported in certain domains, for example, cost estimation models and fault prediction modules.

Chulani et al. (1999) proposed a new approach for using Bayesian analysis in empirical software engineering cost models to predict the cost schedule and quality of the software under development. It compared and contrasted this Bayesian analysis with the commonly used classical multiple regression approach, and concluded that the Bayesian approach was better and more robust than the regression approach.

Menzies and Kiper (2001) addressed the situation in which there is not enough data for software managers to make categorical decisions about software. They proposed a new way that uses stochastic search to generate logs of behavior from the models of subjective knowledge and then applies inductive learning to summarize those logs. By this way, it makes machine learning practical for software engineering problems even in data starved domains.

Stefano and Menzies (2002) performed a case study on software reuse using three different styles of learners: decision tree induction learner, association rule learner and a treatment learner. By comparing the conclusions about the potential success of a reuse program, they concluded that a single learner is not sufficient to find all the applicable patterns buried in a data set, while using a variety of learners can produce a more definitive and useful results.

Zhang (2000) formulated some software development and maintenance tasks as learning problems and applied the corresponding learning algorithms to these tasks. The areas includes: component reuse, rapid prototyping, requirement engineering, reverse engineering, validation, software defect prediction, and so on. It did not give an exact study of a concrete example, but an overview of the possible software engineering areas in which machine learning methods could be helpful.

Our paper discusses a new possible application of machine learning to software engineering areas: change impact analysis. Although we didn’t get satisfying results, it still gives us some hints in applying machine learning algorithms in this field.

## References

- Chulani, S., Boehm, B., & Steece, B. (1999). Bayesian Analysis of Empirical Software Engineering Cost Models, *IEEE Transactions on Software Engineering*, July/August, Vol. 25, No. 4, 573-583.
- Ernst, M. D. (2000). Dynamically discovering likely program invariants (PhD thesis), University of Washington, WA.
- Menzies, T., & Kiper, J. D. (2001). Better Reasoning about Software Engineering activities, *16<sup>th</sup> IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, California, 391-394.
- Ren, X., Shah, F., et al (2003). Chianti: A Prototype Change Impact Analysis Tool for Java (Technical report dcs-tr-533), Rutgers University, NJ.
- Stefano, J. S., & Menzies, T. (2002). Machine Learning for Software Engineering: Case Studies in Software Reuse, *14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)* (pp. 245-251), Washington, DC.
- Zhang, D.(2000). Applying Machine Learning Algorithms in Software Development, *the Proceedings of 2000 Monterey Workshop on Modeling Software System Structures* (pp. 275-285), Santa Margherita Ligure, Italy.