Empir Software Eng (2012) 17:348–389
DOI 10.1007/s10664-011-9180-x

Time variance and defect prediction in software projects

Towards an exploitation of periods of stability and change as well as a notion of concept drift in software projects

Jayalath Ekanayake · Jonas Tappolet · Harald C. Gall · Abraham Bernstein

Published online: 3 November 2011

Editors: Jim Whitehead and Michael Godfrey

Abstract It is crucial for a software manager to know whether or not one can rely on a bug prediction model. A wrong prediction of the number or the location of future bugs can lead to problems in the achievement of a project's goals. In this paper we first verify the existence of variability in a bug prediction model's accuracy over time both visually and statistically. Furthermore, we explore the reasons for such a high variability over time, which includes periods of stability and variability of prediction quality, and formulate a decision procedure for evaluating prediction models before applying them. To exemplify our findings we use data from four open source projects and empirically identify various project features that influence the defect prediction quality. Specifically, we observed that a change in the number of authors editing a file and the number of defects fixed by them influence the prediction quality. Finally, we introduce an approach to estimate the accuracy of prediction models that helps a project manager decide when to rely on a prediction model. Our findings suggest that one should be aware of the periods of stability and variability of prediction quality and should use approaches such as ours to assess their models' accuracy in advance.

J. Ekanayake was partially supported by funding from the Sabaragamuwa University of Sri Lanka while J. Tappolet was partially supported by funding from the Swiss National Science Foundation award number 200021–112330.

J. Ekanayake (✉) · J. Tappolet · A. Bernstein
Dynamic and Distributed Information Systems, Institute of Informatics,
University of Zurich, Zurich, Switzerland
e-mail: jayalath@ifi.uzh.ch

J. Tappolet
e-mail: tappolet@ifi.uzh.ch

A. Bernstein
e-mail: bernstein@ifi.uzh.ch

H. C. Gall
Software Evolution and Architecture Lab, Institute of Informatics,
University of Zurich, Zurich, Switzerland
e-mail: gall@ifi.uzh.ch

Keywords Time variance · Mining software repository · Defect prediction · Decision tree learner · Concept drift

1 Introduction

Many different approaches have been developed to predict the number and location of future bugs in source code (e.g., Khoshgoftaar et al. 1996; Graves et al. 2000; Hassan and Holt 2005; Ostrand et al. 2005; Bernstein et al. 2007)). Such predictions can help project managers to quantitatively plan and steer a project according to the expected number of bugs and their bug-fixing effort. Bug prediction can also be helpful in a qualitative way whenever the defect location is predicted: testing efforts can focus on the predicted bug locations.

Many bug prediction approaches (including the ones cited above) use software evolution (or history) information to predict defects. This information is, typically, collected from software development systems such as CVS or Bugzilla. From this data, file related features (i.e. attributes of source-code files) such as number of revisions or number of authors etc. are extracted.

Those features are then used to train a prediction model. To evaluate such a model, it is given the feature values from another time period and the predicted values are compared with observed ones facilitating the quality assessment of the model. The common downside of these approaches is their temporally coarse evaluations. Usually, a bug prediction algorithm is evaluated in terms of accuracy in only one or a small number of points in time. Such an evaluation implicitly assumes that the evolution of a project and its underlying data are relatively stable over time. But, according to findings of Tsymbal (2004) and Widmer and Kubat (1993) this assumption is not necessarily valid. Therefore, a generalization of such a model is difficult and jeopardizes correct decision-making by software managers.

Given the dynamic nature of software evolution data, the purpose of this paper is to analyze the variability in prediction quality and measure the reliability of prediction models. The general hypothesis of this paper is: *Defect prediction quality varies over time exhibiting periods of stability and variability.* Derived from this hypothesis, this paper focuses on the following research questions:

RQ1 *Can we develop methods to assess the prediction quality over time?*
RQ2 *Is it possible to identify periods of stability in the prediction quality?*
RQ3 *Can we identify elements from the models' input features which are responsible for the variability?*
RQ4 *Can we develop a method to predict the future variability of a prediction model?*

Possible reasons for a high variability could be the sudden change of influencing factors such as a changing number of developers, the use of a new development tool, or even political/economical events (e.g., financial crisis, holiday seasons). Our goal in this work is to uncover possible candidate factors by looking for correlating features in the development process and *provide software managers with a decision procedure to evaluate the prediction model's accuracy in advance*.

Our approach can be summarized as follows: Similar to other techniques described in Kagdi et al. (2007) we use software evolution data to extract file-related features.

The set of features chosen reflects data about the files itself and its history (see Table 2). In addition we extract these features for many time periods of the investigated projects. From this data we then train our prediction models.

1.1 General Overview of Experiments

The first set of experiments, described in Section 5.1, addresses RQ1. First, for one prediction time period—in our case a single month, i.e. the target period—many prediction models are trained using datasets generated from every possible training period.

This procedure is repeated with varying target periods. The predictive power of the models is measured using the receiver operating characteristics (ROC) and the area under the ROC (AUC). For example, if a given software project is evolved over the last 36 months then we use data from the past 35 months to train 34 different bug prediction models and then predict defects on the 36th month. Second, we use the same model to predict defects on every possible target. For example, a prediction model is trained using the data from the first 10 months and then this model is used to predict defects from month 11 onwards until the end of the observed period. To substantiate our claims we illustrate the predictions using heat-maps as a visual tool and additionally employ statistical methods to further support our observations. Our results indicate that there are periods of stability and variability of prediction quality over time and, hence, project managers should not always rely on bug prediction models without such stability information.

Our second set of experiments, described in Section 5.2, addresses RQ2: We first determined a suitable threshold for the prediction quality measure denoting periods of "good" predictions computed by the BugCache algorithm (Kim et al. 2007). Using this threshold we then graphically illustrate how each of the four investigated projects exhibits periods of stability in terms of prediction quality and change.

The third set of experiments is aimed at establishing statistically that the observations about stability and variations are not random. To that end we intersperse our features with random information and show that our experiments are statistically unaffected by the random data and, hence, show a non-random phenomenon.

The fourth and fifth set of experiments, described in Sections 5.4 and 5.5, address RQ3. Using regression analysis we empirically uncover potential reasons for the variability of prediction quality that can serve as early indicators for upcoming variability. For example, we observe that an increasing number of authors editing a project causes a decline in prediction quality. Another observation is that more work done for fixing bugs relative to other activities reduces the prediction quality. Moreover, more authors being active in the training period and fixing many bugs help increasing the prediction quality.

The last set of experiments, described in Section 6, addresses RQ4 to make the results actionable. Using the insights of the previous experiments as a foundation we developed a tool that evaluates the quality of the prediction models. Specifically, we train a meta-model that predicts the quality of the models. Using these 'meta-predictions' software project managers can easily decide when to use bug prediction models and when to forgo them given their (expected) bad quality and/or reduced expressivity.

2 Related Work

Research in mining software repositories investigates the usage of historical data from software projects for various kinds of analyses (as described in Kagdi et al. 2007). One line of this research focuses on building models for the prediction of the occurrence of future defects, changes, or refactorings (cf. Diehl et al. 2009). To put our work in relation to these studies, we discuss a brief selection of related papers. Note, however, that to the best of our knowledge, there is no prior work investigating the possible variation in defect prediction quality over time and its causes.

2.1 General Issues in Bug Prediction

A critical survey of defect prediction models was conducted by Fenton and Neil (1999). They claim that there are numerous serious theoretical and practical problems in many studies. In particular, they mentioned five issues regarding defect prediction models: (1) unknown relationship between defects and failures, (2) problems with the multivariate statistical approach, (3) problems of using size and complexity metrics as sole predictors of defects, (4) problems in statistical methodology and data quality and (5) false claims about software decomposition. In this work, we tried to avoid the above issues. Nevertheless, this was not completely possible, and, therefore, we mention those problems in Section 4 (Threats to Validity). Additionally, to ensure methodological soundness, we employed the methods described in Zimmermann et al. (2007) to link CVS and Bugzilla databases and rely on Eaddy et al. (2008), Antoniol et al. (2008), and Bird et al. (2009) to validate the defect datasets.

Lessmann et al. (2008) introduce a framework to compare defect prediction models. They criticize the usage of only a few number of datasets which might also be proprietary. Furthermore they criticize the usage of inappropriate accuracy indicators and the limited usage of statistical testing procedures to substantiate findings. We are convinced that we address all these issues by (1) using well-known open-source projects as data sources, (2) reporting AUC for measuring the accuracy of our prediction models and (3) using statistical tests where appropriate.

2.2 Different Approaches for Bug Prediction

Apart from our work, there are different approaches that try to predict defects in software systems based on their source code information. For instance Hassan (2009) proposed complexity metrics that are based on the code-change process. He used concepts from information theory to define the change complexity metrics. He considered the code-change introduction process in order to measure the change probability or entropy during specific time periods. Their definition of time frames is similar to the one used in this paper.

Li et al. (2005) present an approach for the prediction of model parameters for software reliability growth models (SRGMs). These are time-based models using metrics-based modeling methods. They used three SRGMs, seven metrics-based prediction methods, and two different sets of predictors to forecast pre-release defect-occurrence rates. Our study also uses time-based prediction models to predict the location of defects. However, we predict defects in every possible time period which allows us to perform a continuous analysis of the bug prediction quality. Further, we use only one prediction model—i.e. class probability estimation models—and only process metrics as predictors. Moreover, our goal is to investigate the variability of defect prediction quality over time as opposed to forecasting defect occurrence rate. Ostrand et al. (2005) and Knab et al. (2006) both used code metrics and modification history to train regression models predicting the location and number of faults in software systems. Zimmermann et al. (2007), in contrast, used only code metrics. They all share the following experimental procedure: first, they constructed several file-level and project-level features from the software history and use those features to train prediction models. Then, the feature values from another time period are computed and the predicted values are compared with observed ones. The common downside of these approaches are their temporally coarse evaluations. Usually, a bug prediction algorithm is evaluated, in terms of accuracy, in only one or a small number of points in time. This renders the generalization of models difficult, as such an evaluation implicitly assumes that the evolution of a project and its underlying data are relatively stable over time. In our study, we also use the software history to compute a set of features and some of our features are similar to these studies. In contrast, however, our feature set reflects almost all the changes to a file in the past. In addition, we evaluate our prediction models throughout the project duration in order to show the variability in prediction quality over time and illustrate the limited "temporal generalizability" of bug prediction models.

Khoshgoftaar et al. (1996), Graves et al. (2000), Nagappan and Ball (2005) and Bernstein et al. (2007) all developed prediction models using software evolution data to predict future failures of the software systems. Mockus and Votta (2000) showed that a textual description field of change history is essential to predict the causes for this change. Further, they define three causes for a change: Adding new features, correcting faults and restructuring code to accommodate future changes (i.e. refactoring). We also use the change history for constructing features, but we predict only faults. We partially base our work on the above mentioned related approaches by adopting some of their presented features.

3 Experimental Setup

Now we succinctly introduce the overall experimental setup. We present the data used, its acquisition method, and the measures used to evaluate the quality of the results.

3.1 The Data: CVS and Bugzilla for Eclipse, Netbeans, Mozilla, and Open Office

The availability of data about multiple development cycles and their possible association to the variation in prediction quality was essential to this study. We therefore selected four open source projects with a particularly long development history (>6 years): Eclipse, Netbeans, Mozilla, and Open Office.

Within each project we considered unique file names with file path and source code file type `*.java` in Eclipse and Netbeans, `*.cpp` in Mozilla and `*.hxx` and `*.cxx` in Open Office during the observed periods of each project. Further, we considered only those files that were not marked as `dead` within the observation

Table 1 Analyzed projects: time spans and number of files

Project	First release	Last release	# Files
Eclipse	2001-01-31	2007-06-30	9,948
Mozilla	2001-01-31	2008-02-29	1,896
Netbeans	2001-01-31	2007-06-30	38,301
Open Office	2001-01-31	2008-04-30	1,847
Total			51,992

Note As starting date we picked the first date at which all projects were under development (i.e. Jan 01)

period. We did not follow file renaming events as this is not a feature supported by CVS.

All data was collected from the projects' Concurrent Versioning Systems (CVS)[1] and Bugzilla.[2] The data from the two sources was then linked using the method described in Zimmermann et al. (2007). Even though the method has been shown to lead to bias in terms of links (see Bird et al. 2009; Bachmann and Bernstein 2009) it has been argued that it is highly unlikely to contain false positives (i.e. links between commits and bugs that should not be there). Consequently, we can say that we predict the presence of bugs (fixes) instead of just commits. Note, that in total we considered 114,186 bug reports from all four projects. The manual verification of such a large number of bug reports would be prohibitively expensive.

We also considered reusing existing and verified datasets.[3] However, given that we needed temporally oriented data for our analysis (organized on a monthly basis) we decided to extract the data from the CVS and Bugzilla databases. We will discuss possible bias in the threats to validity in Section 4.

Table 1 shows an overview of the observation periods and the number of files considered. Moreover, Tables 13, 14 and 15[4] provide detailed descriptions about all components and the number of files of those components. In Eclipse, we consider the core components of the products Equinox, JDT, PDE and Platform available in June 2007. We selected all the components from Netbeans and Mozilla available in June 2007 and February 2008, respectively. For Open Office we only used files from the SW component. This component relates to the product *Writer* being the word processor of the Open Office suite.

3.2 The Data: Features

All features available to the decision tree learner are listed in Table 2. These reflect information about or changes made to a file in the past. All the features, with the exception of the target variable `hasBug`, are computed during the training period of a model. The target variable is computed in the test period of the model. The training

[1] http://www.nongnu.org/cvs/

[2] http://www.bugzilla.org/

[3] E.g., http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/ or http://www.cs.columbia.edu/~eaddy/concerntagger/.

[4] Tables can be found in Appendix A.

Table 2 Extracted variables (features) from CVS and Bugzilla

Name	Description
Versioning system features (per file)	
`activityRate`	Number of revisions per month
`lineAdded`	# of lines added
`lineDeleted`	# of lines deleted
`lineOperationRRevision`	Number of line added and deleted per revision
`revision`	Number of revisions
`totalLineOperations`	Total # of lines added and deleted
Bugtracker features	
`blockerFixes`	# of blocker type bugs fixed
`blockerReported`	# of blocker type bugs reported
`criticalFixes`	# of critical type bugs fixed
`criticalReported`	# of critical type bugs reported
`enhancementFixes`	# of enhancement requests fixed
`enhancementReported`	# of enhancement requests reported
`majorFixes`	# of major type bugs fixed
`majorReported`	# of major type bugs reported
`minorFixes`	# of minor type bugs fixed
`minorReported`	# of minor type bugs reported
`normalFixes`	# of normal type bugs fixed
`normalReported`	# of normal type bugs reported
`trivialFixes`	# of trivial type bugs fixed
`trivialReported`	# of trivial type bugs reported
`p1-fixes`	# of priority 1 bugs fixed
`p1-reported`	# of priority 1 bugs reported
`p2-fixes`	# of priority 2 bugs fixed
`p2-reported`	# of priority 2 bugs reported
`p3-fixes`	# of priority 3 bugs fixed
`p3-reported`	# of priority 3 bugs reported
`p4-fixes`	# of priority 4 bugs fixed
`p4-reported`	# of priority 4 bugs reported
`p5-fixes`	# of priority 5 bugs fixed
`p5-reported`	# of priority 5 bugs reported
Compound features	
`chanceRevision`	Likelihood of a revision in the target period computed using $1/2^i$
`chanceBug`	Likelihood of a bug in the target period computed using $1/2^i$
`lineAddedI`	# of lines added to fix bugs
`lineDeletedI`	# of lines deleted to fix bugs
Project level features	
`lineOperationIRbugFixes`	Average number of lines operated to fix a bug
`lineOperationIRTotalLines`	# of lines operated to fix bugs relative to total line operated
`lifeTimeBlocker`	Average lifetime (avg. lt.) of blocker type bugs
`lifeTimeCritical`	Avg. lt. of critical type bugs
`lifeTimeMajor`	Avg. lt. of major type bugs
`lifeTimeMinor`	Avg. lt. of minor type bugs
`lifeTimeNormal`	Avg. lt. of normal type bugs
`lifeTimeTrivial`	Avg. lt. of trivial type bugs
`totalLineOperationsI`	Total # of lines touched to fix bugs
`grownPerMonth`	Project grown per month (can be negative)
`hasBug` (target)	Indicates the existence of a bug

period is always before the test period with no overlap, as the target variable `hasBug` is only known ex-post (i.e. in the future) whilst the other features are available ex-ante (i.e. at the time where a prediction is made).

To investigate the robustness of the predictions against variation in the training period length we vary its length from two months to the maximum length possible given the data collected.

To determine the correct value for each feature the length of the month is dependent on its calendar length (i.e. starting on the 1st and ending on the 28th, 29th, 30th, or 31st of the month).

Target Feature We consider that a change or a revision is made to a file as a bug-fixing activity if there is a referenced (or linked) entry in the bug database and that is marked as a defect (Zimmermann et al. 2007). In the referenced (or linked) cases the bug database contains information about the opening date of the bugs. Therefore, we can determine the number of bugs that have been reported for each file during a specific time period. When commits do not reference an entry in the bug database then we consider those as non-bug-fixing commits which are all changes that are not related to a defect (i.e. feature enhancements or refactoring activities).

Other Features Most of the names of the features listed in Table 2 are self-explanatory. Some need additional context, which is provided below.[5]

The `activityRate` represents how many activities (revisions) took place per month. To determine the rate we count the number of revisions during the training period and then divide it by the length of the training period (leading to averaging of the value). The length of the training period is given by months.

The features `lineAdded` and `lineDeleted` are the total number of lines of code added and deleted in all revisions during the training period. The sum of these two features is `totalLineOperations`.

`grownPerMonth` describes the evolution of the overall project (in terms of lines of code) in the training period. Specifically, we compute the difference between number of lines added and deleted. This number can be positive (growth) or negative (shrinkage). We then average this value by dividing it by the length of the training period (in months).

The feature `lineOperationRRevision` describes the average number of lines added and deleted per revision during the training period.

The `chanceRevision` and `chanceBug` features describe the probability of having a revision and a bug in the future as used in the *BugCache* approach (Kim et al. 2007) discussed in Section 2. We compute those two features using the formula $1/2^i$, where i represents how far back (in months) the latest revision or bug occurred from the prediction time period. If the latest revision or bug occurrence is far from the prediction time period, then i is large and the overall probability of having a bug

[5]Note that a complete description can be found in Appendix B and that for all features where authorship is relevant it is determined as the person committing the code into the CVS rather than the developer noted in the comments of the code. However, most of active contributors are committers of a project. For example in the PDT project (http://www.eclipse.org/pdt/people/contributors.php#Seva-%28Wsevolod%29-Lapsha), out of 12 participants 11 of them are committers. Hence, this assumption will not have a great impact on the outcome of the experiments.

(or revision) in the near future is low. Hence, these variables model the scenario that files with recent bugs are more likely to have bugs in the future than others (see Kim et al. 2007; Hassan and Holt 2005).

The features from `blockerFixes` to `p5-reported` provide information about the different types of bugs reported and fixed for files during the observed training period. If an opening date of a bug reported for a file falls into the training period then a bug is considered as being reported during the training period. Analogously, we count the number of fixed bugs.

`lineAddedI`,`lineDeletedI` and `totalLineOperationsI` provide the number of lines operated to fix bugs and `lineOperationIRbugFixes` provides the average number of lines operated per bug during the training period.

`LineOperIRTolLines` counts how many lines were added and deleted to fix bugs in relation to the total number of lines added/deleted. This indicates what fraction of changes is focused on fixing bugs in relation to other activities (such as adding new features).

Finally, the remaining features represent the average lifetimes of bugs. Note that if a fixed bug is revised then the revision date is considered as the closing date of that bug. The corresponding entry for the bug fixing revision in the bug database provides the opening date of the bug and, hence, we can compute the lifetime of the bug. Though the opening date lies outside the training period we still use it to compute the lifetime of the bug.

3.3 Performance Measures

In our experiments we train two types of models: class probability estimation and regression models.

For most of our experiments we trained class probability estimation (CPE) models. Specifically, we used a simple decision tree inducer: Weka's (Witten and Frank 2005) J48 decision tree learner. This is a reimplementation of C4.5 introduced by Quinlan (1993), which predicts the probability distribution of a given instance over the two possible classes of the target variable: `hasBug` and `hasNoBug`. Typically, one then chooses a cut-off threshold to determine the actual predicted class, which in turn can be used to derive a confusion matrix and the prediction's accuracy. We introduce misclassification cost when training a model such that both misclassification costs—false negative and false positive—are equal.

Our datasets have a heavily skewed distribution, i.e. the ratio between defective files and non-defective ones is, depending on the project, about 1:20 and approximately keeps this ratio in all samples. For that reason we do not use the confusion matrix and associated accuracy as our performance measure as they are heavily influenced by this prior distribution. Instead we use the receiver operating characteristics (ROC) and the area under the ROC curve (AUC), which relate the true-positive rate to the false-positive rate and is independent of the prior distribution (see Provost and Fawcett 2001). An AUC of 1.0 represents perfect, and 0.5 random prediction quality.

For the regression experiments we use linear regression models. The linear regression is a form of regression analysis in which the relationship between one or more independent variables and another variable, called the dependent variable, is modeled by a linear function that minimizes the squared error of the weights associated with the independent variables. This function is a weighted linear combination of one or more model parameters, called regression coefficients. We report Pearson correlation, root mean squared error (RMSE), and mean absolute error (MAE) to measure the performance of the regression models.

4 Threats to Validity

In this section we briefly discuss the most important threats to validity concerning the data gathering process, the data itself, and the applied methodologies.

4.1 Determination of Authorship

Throughout this paper, we consider the committer to be the author of a change. This is possibly wrong when looking at projects that do not allow direct write-access to CVS. Apache's code base, e.g., can only be changed via trusted proxy persons (i.e. committers). For our experiments, we did not consider source code information and therefore needed to rely on the information available from the versioning system (CVS). Even with the source code information available (e.g., relying on the `@author` tag from Javadoc) we could not be sure that the listed person is also the author of the code change. Consider the following example: a developer makes a minor addition to the code in order to fix a defect and does not add himself to the list of authors in the source code because he thinks it is not worth mentioning. In such a situation the initial author of the file would be considered to also have made the bug-fix. Hence, our method is limited to determining the person who brought the code into the project's codebase—but that without doubt (be it as the actual author or not).

4.2 Creation-Time vs. Commit-Time

In this work we did only consider data that is made publicly available by the developer. Since we use a time-based partitioning of the datasets we make an implicit assumption that bugs occur at the moment when they are reported and are being fixed at the moment when a respective code change is committed. This may not always be correct because a code change may have been made long before committing (on the developers private workspace). Also, a bug might be in the code for months (or years) without being noticed. Given the available data we see no way to address this limitation. However, from a project management perspective it can be argued that defects and code changes only become relevant when they are reported. Only at reporting time they "materialize" as a task for the development team and cause further actions.

4.3 Bug-Fixing or Enhancement? A Clear Case of Bias

It is hard to distinguish between bug fixing efforts and enhancements of the code (e.g., the addition of new features or refactoring). Oftentimes, developers make a connection between a bug report and its related code changes by mentioning a bug ID in the commit message. However, this is a brittle connection without any mechanisms granting exclusivity of the submitted files to the mentioned bug report. A clear distinction between bug-fixing and code enhancement activities would require manually verified datasets (see also Bachmann and Bernstein 2009). In addition to the brittle connection some information could be outright missing. For instance, a minor bug that is quickly fixed changes its state from open to fixed; without having a priority and/or severity assigned. Consequently, the data of this study clearly exhibits both commit feature bias as well as bug feature bias as introduced by Bird et al. (2009). In addition, Ko and Chilana (2010) revealed that most of power users reported non-issues that devolved into technical support, redundant reports with little new information and expert feature requests, and the reports that did lead to changes were reported by a comparably small group of experienced, frequent reporters. This implies that even the power users have no clear intention about the state of the reports.

Unfortunately, barring the availability of manually verified models, we we see no practical way to address these biases. A main characteristic of the methods used in this work is the long-term evaluation of prediction models on software projects. To manually verify our datasets we would need to look into every bug report and every code change of a whole project and its history—an effort clearly beyond the scope of this study.

4.4 Choice of Time Frames

We chose two-month windows as datasets for our prediction models. We do not insist that this window is the only or even correct one. We decided on two months because it was a time-frame that we found useful in one of our previous studies (see Bernstein et al. 2007) and is a release cycle that we observed in different projects. Often, a version/milestone is reached after 6–8 weeks. Obviously, software projects, just like any other projects, can exhibit some form of entrainment (see Ancona and Chong 1996). For future work it would be interesting to (i) assess the entrainment cycles[6] and (ii) investigate the robustness of our results when narrowing the time windows to, e.g., days or weeks.

4.5 Choice of Observed Periods of Projects

We selected January 1, 2001 as the starting date of all four projects. At this date all projects were under development. However, at this date not all projects were very mature and, hence, their data might be inconsistent. Eclipse and Open Office, for example, had no bug reports during a period of nine months in 2001; this is probably due to the early state of the projects or their less systematic use of a bug-tracker, and not because these projects were bug free. Nonetheless, as our analyses will show, the starting date should not have an undue influence on our results, as we investigated long time periods (>6 years).

[6]E.g. how the individual committers coding behavior synchronizes towards a milestone.

5 Experiments: Change in Bug Prediction Quality

In this section we explore the nature and possible causes for the variation in bug prediction quality. From these findings we finally develop a "meta-model" that predicts the quality of the prediction models in advance. This meta-model helps project manages to decide when to use their bug prediction model and when not—a goal we explore in Section 6.

5.1 Defect Prediction Quality Varies Over Time

The first research question we explore is whether defect prediction quality varies over time. To that end, we conduct two different kinds of experiments. In the first experiment, we keep the target period constant and predict defects on that target using the models trained on data collected from every possible combination of training periods. In the second experiment, we keep the prediction model constant and predict defects on varying target periods. As mentioned we use Weka's (Witten and Frank 2005) J48 decision tree learner as a CPE induction method, which is trained with the features listed in Table 2. In both experiments the algorithm predicts the location of defects: i.e. it predicts which files will (or will not) contain bugs in the target period. The datasets for these two experiments is described in Appendix C.

For the first experiment, we start predicting defects in the last month—the target period—of the observed period of each project using the models trained from data collected in two months—the training period—before the target period. Next, we expand the training period by one month in order to collect more information still predicting on the same target period. This procedure is repeated until the training period reaches the maximum possible length into the past. Consequently, the maximum length for the training period in Eclipse and Netbeans is 74 months, for Mozilla 82 months, and for Open Office 85 months. Then, we move the target period one month backwards and repeat the above procedure. For example, if the initial target period is the month t and the initial training period is $[t-1, t-2]$, followed by $[t-1, t-3]$ etc. Next we move the target period to $t-1$ and the initial training to $[t-2, t-3]$ and repeat the procedure. For each training run we measure the model's prediction quality using its AUC value and visualize it using a heat-map. Since all the projects show similar characteristics we only show the results for Eclipse in Fig. 1. In this heat-map, the X-axis indicates the target period and the Y-axis the length of the training period in terms of number of months (i.e. for the training period $[t-n, t-m]$ Y is $m-n$).

We further compute the maximum, minimum, mean, and variance of the AUC values as well as the histogram of AUC variance's in each *column* of the Eclipse heat-map (cf. Fig. 1) and visualize them using bar charts as in Fig. 2. In the bar charts (Fig. 2a–d), the x-axis shows the target period and the y-axis shows the AUC. In Fig. 2e, the x-axis shows the bin values and the y-axis shows the frequencies. According to the observations in Fig. 1, the models involved in predicting defects on certain target periods (e.g., April 2005) obtain an AUC around 0.9 (see Mean AUC in Fig. 2c) while the models that predict defects in August 2003 obtain an AUC around 0.6. In some prediction periods (e.g., March 2006) the prediction quality is initially relatively low but when expanding the learning period up to certain months back the models gain prediction quality. In other cases (e.g., July 2005), in contrast, a further

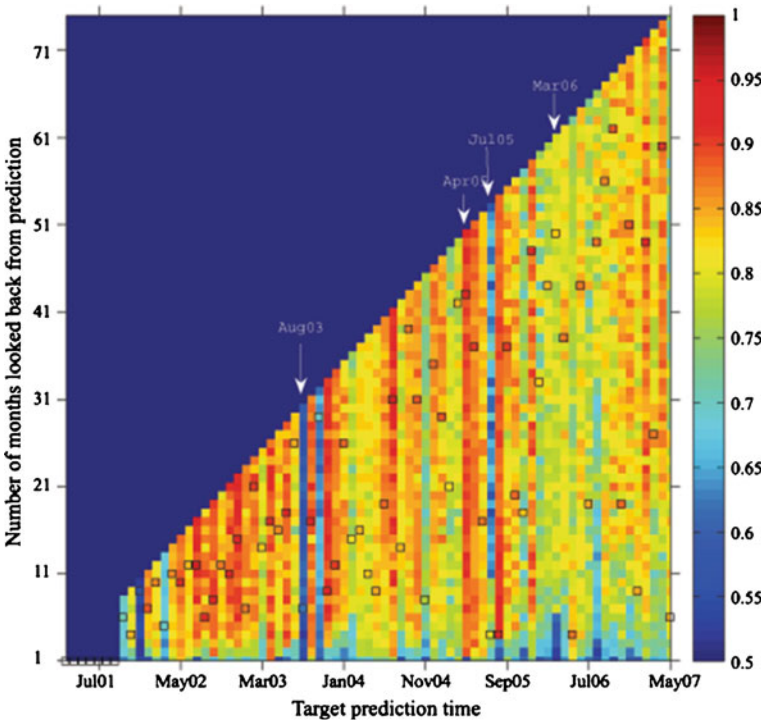


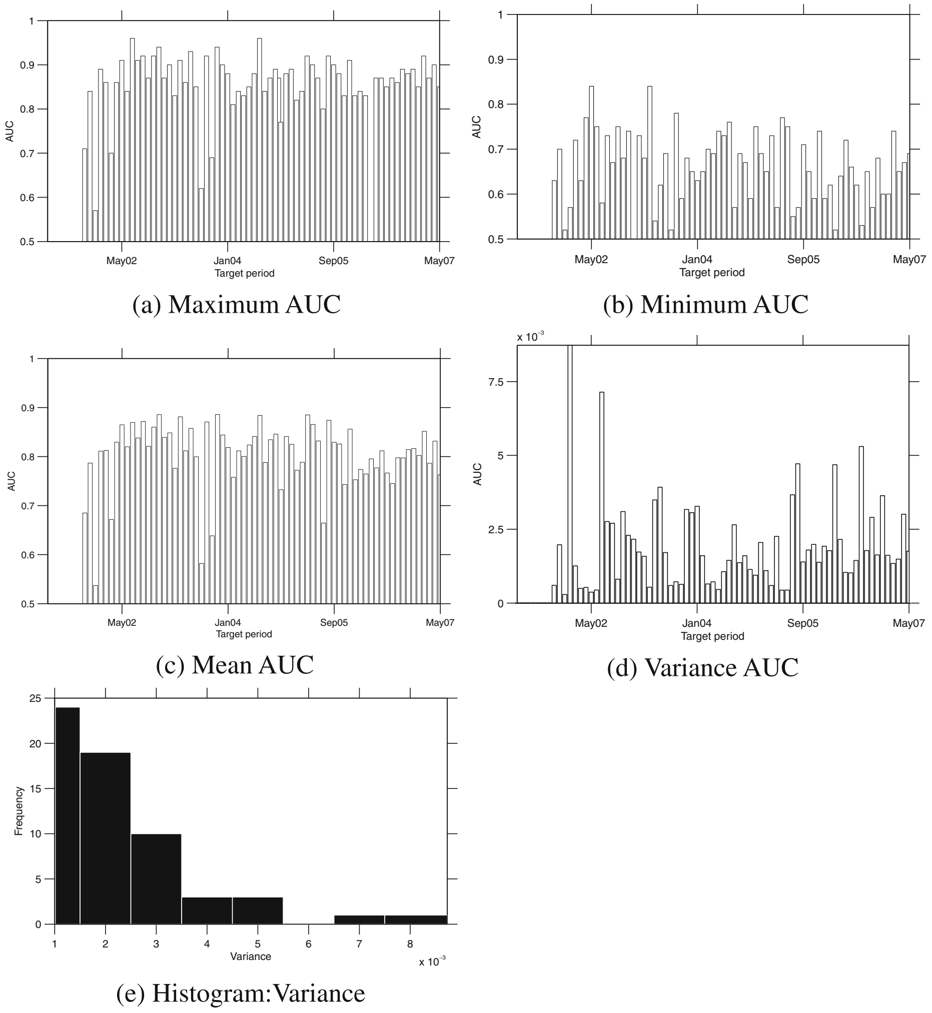
Fig. 1 Eclipse heat-map: Prediction quality using different training periods with the points of highest AUC highlighted

expansion of the training period causes a degradation of prediction quality. The maximum AUC values for each target period (shown as square in Fig. 1) typically lie on neither ends. This suggests that in order to obtain higher prediction accuracy, the models should not be trained on data collected from a very long or very short history.

Hence, models at different time periods (and varying length) seem to vary, but is the variation significant? To establish that the AUC varies significantly we compare the distribution of high and low AUC-variance values. Specifically, we use the split-half method as described by Ko and Chilana (2010): First, we rank the variance values in descending order and divide them into two equal parts. Second, we compare the higher and lower parts using the Mann–Whitney signed rank test (at $\alpha = 95\%$) and find that the mean rank AUC-variance is significantly different (see Table 3 for details).[7]

What we have learned so far is that when we keep the target constant but vary the training then we find some variation in prediction quality. In the second experiment

[7]Note that we used the Mann–Whitney test as the test for normality (one-Sample Kolmogorov–Smirnov test: $p = 0.055$) produced a borderline result. As some still use the t-test for large collections of slightly non-parametric data we also ran an independent-sample t-test and found it to be significant at $\alpha = 0.001$.



(a) Maximum AUC (b) Minimum AUC

(c) Mean AUC (d) Variance AUC

(e) Histogram:Variance

Fig. 2 Descriptive statistics of AUC values in each *column* of the Eclipse heat-map (Fig. 1)

we establish that the prediction quality also varies when we keep the training period constant and change the target.

To that end our second experiment initially trains a prediction model using the data collected from the first two months of the observed period and then uses this model to predict defects on the third month, fourth month until the last month of

Table 3 Half-split test on *column* AUC values ($P = 0.00$ at 95% confidence level): keep target constant and varying training length

	# of values	Mean rank
Upper half	34	51.5
lower half	34	17.5

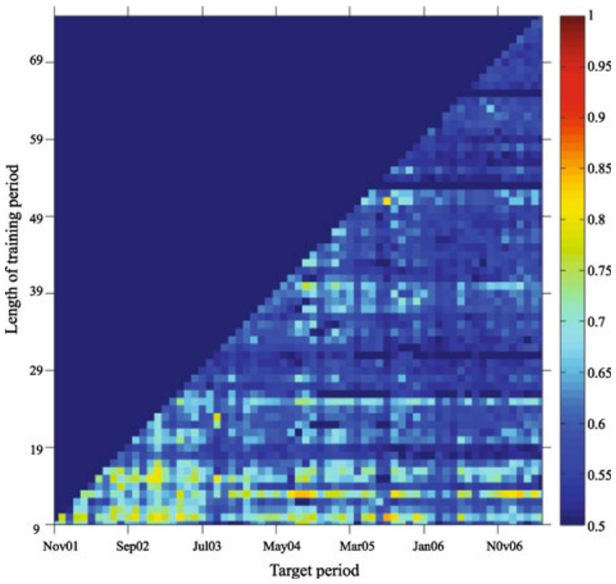
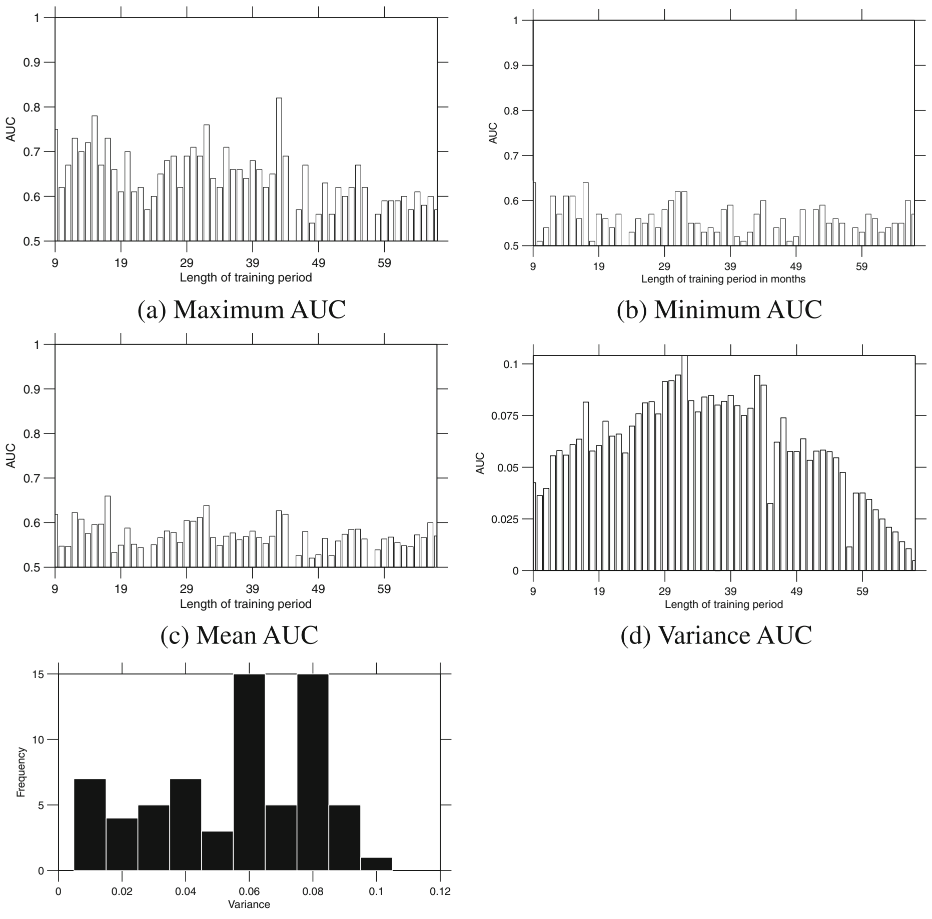


Fig. 3 Eclipse heat-map: Prediction quality at different target periods

the observed period. Next we expand the training period by one month and start predicting defects from the fourth month onward. This procedure is repeated until the training periods reach the maximum observation period. Similar to the first experiment, we measure model's prediction quality for each target period using AUC value and visualize all of these values in a heat-map (see Fig. 3; x- and y-axes are the same as in Fig. 1). In Fig. 3 the bottom row of the heat-map, for example, shows the AUCs of the models trained from the data collected from the first two months of the observation period. The target periods are the third, fourth and so on until the last month. The second row (from the bottom) shows the AUCs of the models trained from the first three months of the observation period and the target periods are from the fourth month onward. Analogously we compute the descriptive statistics of the AUC values in each *row* of this heat-map and visualized in bar charts as in Fig. 4. In the Fig. 4a–d, the x-axis shows the length of the training period and the y-axis shows the AUC values. The x- and y-axes of bar chart (Fig. 4e) is the same as in the above experiment. Following the one-sample Shapiro–Wilk test for normality ($p = 0.442$) we repeat the split-half method on the variance values of AUC as above and compare the high and low AUC-variance values using the Mann–Whitney signed rank test (at $\alpha = 95\%$) as shown in the Table 4.[8] Again we find significant variation over time. Note, however, that this experiment does not address the question of establishing the optimal training period—a question we leave open for future work. It is also important to note that the models trained in different periods are likely to rely on different combinations of predictors.

[8]Like above a t-test reconfirmed these findings at $\alpha = 0.001$.



(a) Maximum AUC

(b) Minimum AUC

(c) Mean AUC

(d) Variance AUC

(e) Histogram: Variance

Fig. 4 Descriptive statistics of AUC values in each *row* of the Eclipse heat-map (Fig. 3)

Summarizing, these two experiments show that the prediction quality varies over time: both when holding the model constant and predicting varying target periods (change along the *x*-axis in Fig. 3) as well as when sliding the training period while predicting the same target (change along the *y*-axis in Fig. 1). Hence, models that are good predictors in some target periods are likely to be bad ones in others and the prediction quality of models for a given target period varies based on the training period.

Table 4 Half-split test on *row* AUC values ($P = 0.00$ at 95% confidence level): keep model constant and varying target

	#of values	Mean rank
Upper half	33	63.0
lower half	34	21.5

5.2 Finding Periods of Stability and Change

So far we have seen that the prediction quality varies over time. Hence, it is worth investigating whether there are periods when the prediction quality is good and this trend continues and forms a period of stability or when there are periods of continuous changes of the model's prediction quality.

To differentiate periods of stability and change we slightly adapted our experiment as follows: similar to the first experiments we kept the target period constant but varied the training period. In contrast, we used a two-month training window and slided this training window into the past. The format of the dataset is the same as described in Appendix C, but the difference is that the length of the training period is two months. We decided to use a two-month training window because the release cycle of the considered projects is typically 6–8 weeks. In addition, the work of Bernstein et al. (2007) has shown that two months of history data attains higher prediction quality. Again, we employed Weka's J48 decision tree learner.

Figures 5, 6, 7 and 8 visualize the results of this procedure for the considered projects. Note that whilst the X-axis of these graphs shows the target period as before, the Y-axis has a different meaning: it represents the time-difference between the target period and the two-months training window in months. Hence, the higher in the figure we are looking the older the two-month period is compared to the

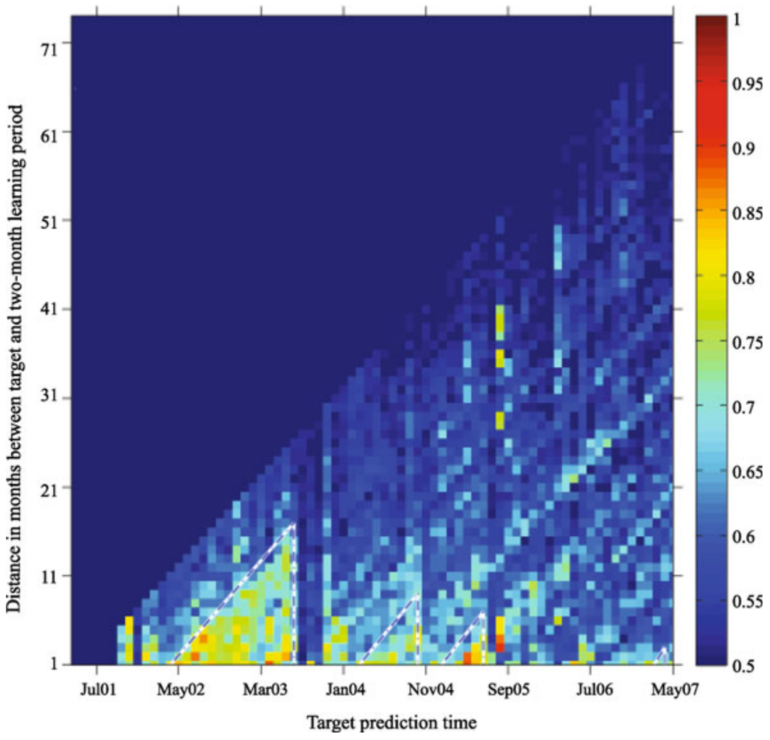


Fig. 5 Two-month Heat-map: Eclipse. *Note* In the first nine months there are no bug reports in the target period and therefore no prediction model was built (*white area* at the *bottom left* corner)

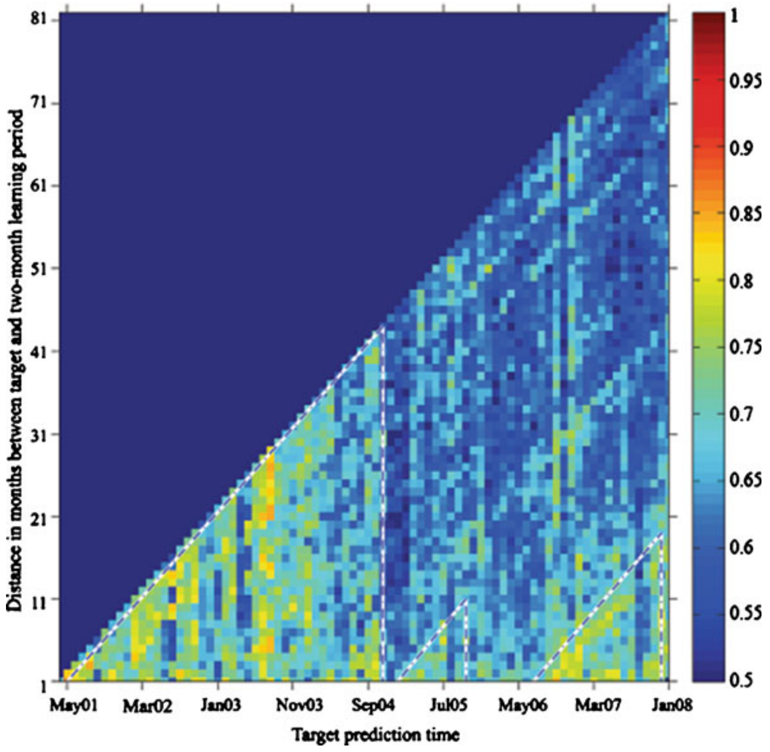


Fig. 6 Two-month Heat-map: Mozilla

target. Values on the diagonal (bottom left to top right) from each other represent predictions of the model trained on the same period.

By looking at the the above heat-maps we can see some triangle shapes (**dark color**). For instance, in Fig. 5, one such triangle starts from April 2002 and continues till July 2003. During these periods the prediction quality stayed relatively stable and a triangle seemingly emerges as the old training data (along the upper left boundary/diagonal of the triangle) remains predictive. But what is a "good" prediction quality?

To identify periods of stably good predictions and maintain that the triangles are indicative of these periods we need a notion of what "good" predictions are. Whilst the AUC scale clearly has some boundaries for "perfect" (= 1) and "random" (= 0.5) it is not necessarily clear what can be regarded as "decent" in any particular task.

To determine a notion of "decent" for our task empirically we first determined the attainable prediction quality on our data using the BugCache[9] prediction model by Kim et al. (2007). We ran BugCache on our Eclipse project data (the observed period is from April 2001 to January 2005) with different cache sizes (as the number of files varies) and present the results in Table 5. The table shows the minimum, maximum,

[9]More precisely, we used FixCache as BugCache is only the theoretical model behind the method. Nevertheless, BugCache is the often-used term for both methods.

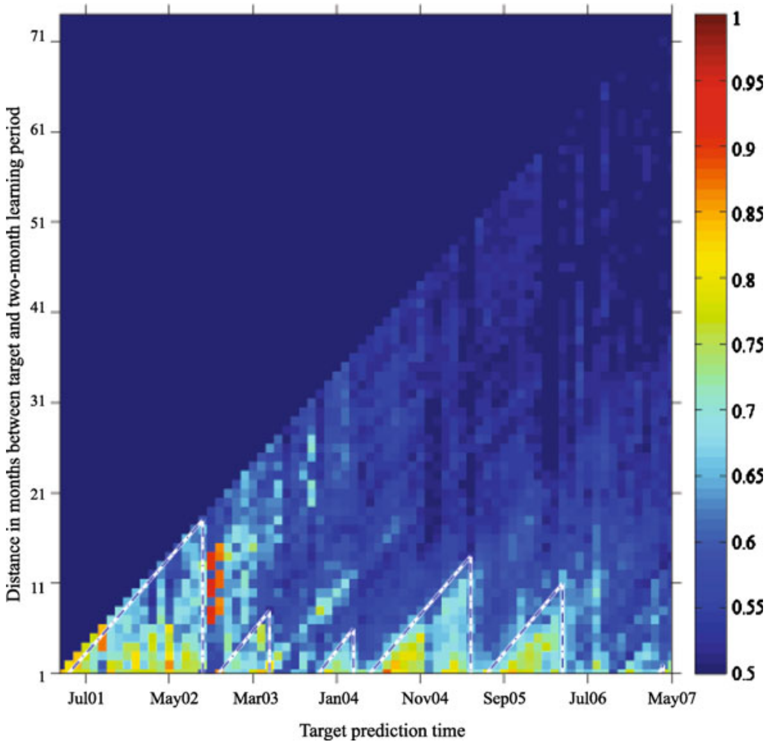


Fig. 7 Two-month Heat-map: Netbeans

median and mean of AUC for each of the runs. As the table shows the maximum attained AUC varies between 0.66 for the smallest cache and 0.76 for the largest one. Given BugCache's usual prediction quality we decided to take the lower end of these values as indicating "sufficiently good" and set our threshold for "decent" predictions on our data to AUC = 0.65.

To illustrate the resulting triangle shapes Figs. 5–8 indicate periods where more than 80% of the values are higher than the threshold with a drawn triangle. Consequently, we find that the prediction periods inside the triangles are stable even on models learned from older data. Returning to the stable example period in Eclipse (April 2002–July 2003 in Fig. 5) we find that even data from the second quarter of 2002 (more than a year old) provides a decent prediction quality.

Table 5 Prediction quality (AUC) of the model (Kim et al. 2007) for Eclipse project; observed period is Apr 2001–Jan 2005

Cache size%	min	max	Median	Mean
10	0.45	0.66	0.51	0.52
15	0.40	0.69	0.51	0.52
20	0.40	0.74	0.52	0.54
25	0.40	0.76	0.55	0.57

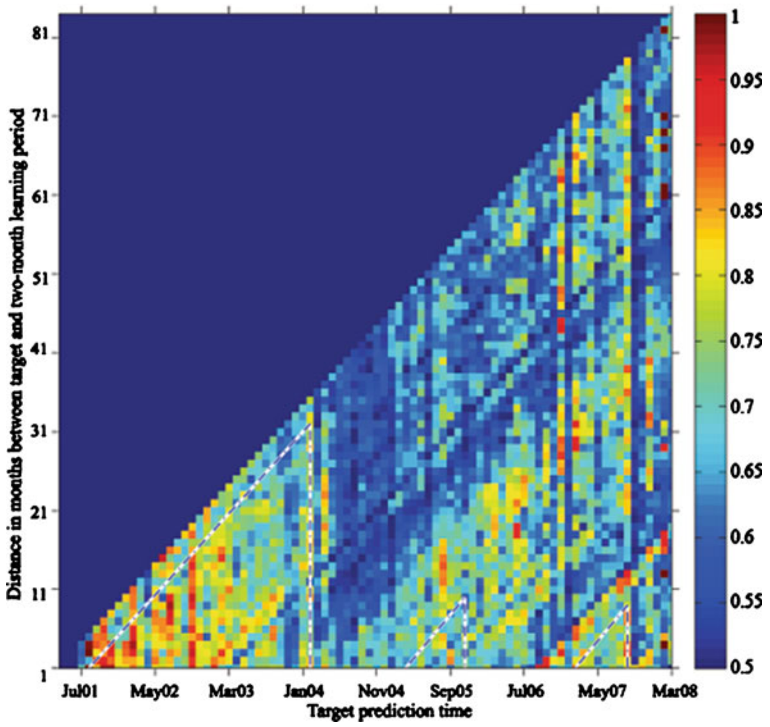


Fig. 8 Two-month Heat-map: Open Office. *Note* In the first four months there are no bug reports in the target period and therefore no prediction model was built (*white area* at the *bottom left* corner)

In all figures we also observe that the further we move the training period to the past the more likely the prediction quality would drop down to almost random (≈ 0.5). This provides some evidence to the statement *the further back you go in time the more the prediction deteriorates* (Kenmei et al. 2008). More formally, from April 2002 to July 2003 the model exhibits a stable good prediction quality. In March 2004 the project seems to recover some stability in defect prediction quality and generate another, but slightly less pronounced triangle until October 2004. The triangles exhibited by the Netbeans project look similar to the one of Eclipse: relatively small (approximately 1 year) but with a high frequency. Mozilla and Open Office, on the other hand, have long periods of stability (e.g., Mozilla: from May 2001 until November 2004). In such a period, a two-month training window, which is older than three years can predict defects with decent accuracy of AUC around 0.7.

Summarizing, the model exhibits periods of stability and variability in defect prediction quality over time. The causes of the changes—be they observable in our features or not—are not obvious from the graphs and will be investigated in Section 5.4. Another interesting observation in the heat-maps is the height of the triangle-shapes. This height indicates the length of the stable period. Note that the height varies both within and between projects. Hence, a universal optimal training period length cannot be determined but is highly dependent on the causes for the current stable period. Finally, this finding indicates that decision makers in software project should be cautious to base their decisions on a generic defect prediction model. Whilst they might be useful in periods of stability they should be ignored in periods of variability.

All of these findings assume that the triangle shapes observed are indeed a feature of the underlying software projects rather than a random artifact—a question to which we turn next.

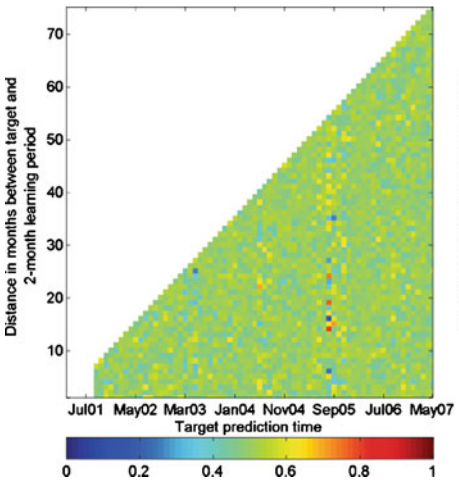
5.3 Triangle Shapes are not Random Phenomena

To illustrate that the triangle shapes are not an epiphenomenon of the data or the prediction algorithm, we graphed the result of a naïve model. In the naïve model we simply assume that if a file in the target period was recorded to contain at least one bug in the given two-month training period then we predict that file is going to be buggy in the target period. Figure 9a shows for Eclipse that most predictions attained in this manner are random (i.e. AUC ≈ 0.5; white in the figure) and do not exhibit the triangle shapes.

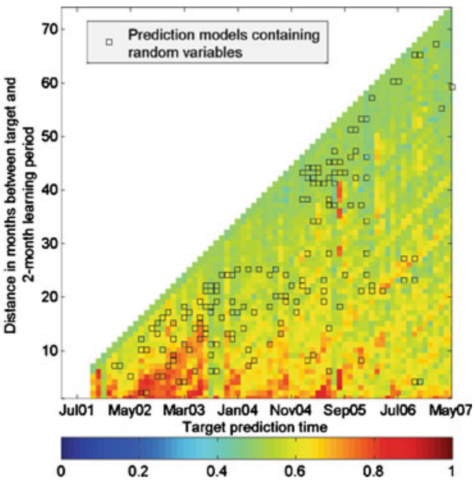
To elicit whether the triangles indeed visualize a phenomenon of the underlying data rather than the prediction process itself, we added ten random variables to our feature set. The random features are generated from similar distributions as ten real variables that are selected randomly. To avoid an outlying result we repeated this procedure four times. In each of these runs the number of models that actually picked up the random features was between 158 to 162 of the total 2,850 models computed.[10] Figure 9b shows the run with 162 models picking random features; the other runs look almost identical. It is important to note that the models containing random features (marked with a square in the figure) are mostly found when the AUC is close to 0.5 (i.e. random). In 14% of the cases does a model with AUC > 0.65 pick up a random variable. Hence, we can assume that they are mostly picked due to the noise in the data. Models pick a maximum of two random features, which appear lower than the 3rd level in the decision tree. Hence, the random features seem to be seldomly used by predictive models (where AUC $\gg 0.5$) and do not seem to be dominating in those models. But do they deteriorate the predictive quality of those models?

To show that the random features have no statistically significant effect on the prediction quality we compare the prediction quality of the models with the random features (see Fig. 9b) to the ones without (see Fig. 5). To that end, we first determine triangles in Fig. 9b (with random feature) using the same method described in the previous section. Further, we confirmed that these triangles are located in the same places as in Fig. 5. We then generate pairs of AUC values as follows: We pick one AUC value from Fig. 5 and the other AUC value from Fig. 9b at the same coordinate. Having confirmed that the data does not follow a standard distribution using visual techniques, we performed Wilcoxon signed rank test comparing those selected values and found them to be significantly similar ($p = 0.446$).

[10]Note that the observed number of models (162) that pick random features is significantly different from the expected number of models (1,425) according to a χ^2-test ($p < 0.001$).



(a) Naïve prediction model



(b) Usage of random features

Fig. 9 Experiments to exclude the possibility of the triangles being an epiphenomenon of the data or the prediction algorithm (Eclipse data)

The above experiments show that our feature set is better than a set of random features and the quality of prediction models (i.e. the ones within the triangles) is not significantly different when random features are present or not. Hence, the triangles must be a result of the underlying models' predictive power given the available data. Having found that our observation of periods of stability and variability is sound we turn to trying to identify the causes for such variability and set the stage for making these observations actionable.

5.4 Finding Indicators for Prediction Quality Variability

In Section 5.2 we show that defect prediction models exhibit periods of stability and change. Can we uncover reasons for such variability?

To that end we learned a regression model to predict the AUC of the bug prediction model according to the following procedure:

First, we computed the AUC of the bug prediction model based on the data in the three months (two months training period and one month labeling period as described in Appendix C) before the target period in exactly the same way as in the previous subsection. The AUC is derived using the file-level features but is a feature of predicting defects within a project. Hence, it is a project-level feature.

Second, since the AUC is a project-level feature, we needed project-level features to train a prediction model. Thus, we computed a series of project-level features (listed in Table 6) by aggregating the respective file-level features in two-months training windows and one-month labeling periods. However, we do not include the information about the labeling period in the features since it is unique and consequently not a useful predictor. We also exclude the distance between the target

Table 6 Project level features for regression

Name	Description
`revision`	Number of revisions
`grownPerMonth`	Project grown per month
`totalLineOperations`	# of lines added and deleted
`bugFixes`	# of bugs fixed (all types)
`bugReported`	# of bugs reported (all types)
`enhancementFixes`	# of enhancement requests fixed
`enhancementReported`	# of enhancement requests reported
`p1-fixes`	# of priority 1 bugs fixed
`p2-fixes`	# of priority 2 bugs fixed
`p3-fixes`	# of priority 3 bugs fixed
`p4-fixes`	# of priority 4 bugs fixed
`p5-fixes`	# of priority 5 bugs fixed
`p1-reported`	# of priority 1 bugs reported
`p2-reported`	# of priority 2 bugs reported
`p3-reported`	# of priority 3 bugs reported
`p4-reported`	# of priority 4 bugs reported
`p5-reported`	# of priority 5 bugs reported
`lineAddedI`	# of lines added to fix bugs
`lineDeletedI`	# of lines deleted to fix bugs
`totalLineOperationsI`	# lines operated to fix bugs
`lineOperIRbugFixes`	Average (avg.) # of lines operated to fix a bug
`lineOperIRTotalLines`	# of lines operated to fix bugs relative to total line operated
`lifeTimeIssues`	Avg. lifetime of bugs (all types)
`lifeTimeEnhancements`	Avg. lifetime of enhancement type bugs
`author`	# of authors
`workload`	Avg. work done by an author
`AUC` (target)	Area under ROC curve

period and the two-months training window since the finding *the further you go back in time worst will be the prediction* is not novel for the software engineering community.

Third, since (i) the AUC prediction model used a two-months training period and (ii) we are interested in changes between the training and the labeling period we transformed the features by taking the average of the two training months ($avg_t = average(feature_{t-1}, feature_{t-2})$) and subtracting it from the values of the labeling month ($feature_t - avg_t$).

Fourth and last, we trained a traditional linear regression model predicting the AUC from these transformed features.

The resulting regression models are shown in Tables 7, 8, 9, and 10.

If a regression coefficient is large compared to its standard error, then it is probably different from zero. The *p*-value of each coefficient indicates whether the coefficient is significantly different from zero such that if it is ≤ 0.05 (with 95% confidence interval) then those variables significantly contribute to the model. In the Eclipse and Mozilla regression models the *p*-value of each coefficient listed in Tables 7 and 8 is better than the standard level of 95%. We did not list coefficients that are not significant. The performance of the models as measured in terms of their Pearson correlation, Spearman's rank correlation, mean absolute error (MAE), and root mean square error (RMSE) is shown in Table 11. Note that all models have a

Table 7 Eclipse: regression model (*p*-value is better than the standard level 95%)

Feature	Unstand	Standard
(Constant)	0.6700	
enhancementFixes	0.0002	0.168
enhancementReported	0.0001	0.125
p1-fixes	−0.0013	−0.494
p3-fixes	−0.0002	−0.481
p5-fixes	−0.0430	−0.071
p1-reported	0.0015	0.540
p2-reported	0.0001	0.118
p3-reported	−0.0001	−0.090
p4-reported	−0.0005	−0.069
p5-reported	−0.0050	−0.145
LineOperIRbugFixes	−0.0010	−0.264
LineOperIRTolLines	−0.1127	−0.244
author	−0.0065	−0.324

moderate correlation between the predicted and actual values of AUC. The small MAE and RMSE reflect the good performance of our regression models.

In all regression models the change in the *number of authors* feature (for brevity we call this `author` in the tables) has a negative impact on the AUC: if the number of authors in the target period is larger than the number of authors in the learning period then the defect prediction quality goes down and vice versa.

Hence, the increase of authors in a project reduces the applicability of the defect prediction model learned without those authors. One possible reason could be that adding more authors to a project may lead to a change in the underlying development patterns in the software project.

The regression coefficients (unstandardized) for `author` in all four models are very small, but since the AUC moves in the range of 0–0.9 they contribute about 1% to the model having at least an indicative character. For example in Eclipse: including

Table 8 Mozilla: regression model (*p*-value is better than the standard level 95%)

Feature	Unstand	Standard
(Constant)	0.7333	
revision	−0.0001	−0.600
bugFixes	0.0001	0.722
enhancementFixes	−0.0012	−0.264
enhancementReported	−0.0004	−0.098
p1-fixes	0.0004	0.153
p2-fixes	0.0003	0.221
p3-fixes	0.0003	0.150
p4-fixes	0.0012	0.185
p5-fixes	−0.0016	−0.393
p3-reported	0.0007	0.202
p4-reported	0.0005	−0.073
p5-reported	0.0010	0.087
LineOperIRbugFixes	0.0011	0.396
LineOperIRTolLines	−0.2478	−0.220
author	−0.0007	−0.137

Table 9 Open Office: regression model

Feature	Unstand	Standard	*p*
(Constant)	0.6700		0.000
bugFixes	−0.0025	−0.034	0.000
enhancementFixes	−0.0022	0.060	0.015
patchFixes	−0.0020	0.056	0.010
featureFixes	−0.0024	−0.178	0.000
enhancementReported	−0.0001	−0.208	0.000
patchReported	0.0001	0.041	0.024
p2-fixes	0.0005	0.176	0.004
p2-reported	0.0025	−0.067	0.038
p4-reported	0.0022	−0.310	0.000
p5-reported	0.0035	0.122	0.000
LineOperIRTolLines	−0.0491	−0.180	0.000
author	−0.0008	−0.037	0.103

ten authors more in the target period than in the learning period will decrease the AUC by 0.065 and this is a considerable amount of decrease in AUC. For the other projects, the influence is an order of magnitude smaller but still pointing in the right direction. For Open Office, the effect is statistically insignificant at the 98% confidence interval. Hence, the effect is observable in most projects but is most pronounced in Eclipse.

Another interesting feature of the models is the number of lines added/removed to fix bugs relative to the total number of lines changed (called `LineOpeIRTotLines`). This feature reflects the fraction of work performed to fix bugs relative to the total work done. In all models except for Netbeans this factor has a high impact compared to the other features. In Eclipse, Mozilla, and Open Office, this factor contributes negatively to the model, while in Netbeans it contributes positively but not significantly.

Hence, if the coefficient is negative (as in Eclipse, Mozilla, and Open Office) then an increased bug fixing activity (compared to new feature additions) will have a negative impact on the AUC—presumably as an increased overall bug fixing effort will increase general code stability and, however, change the relationship between the project level features and the prediction quality of a bug prediction model. In addition, more bug fixing effort will result in changing the underlying defect generation rules and consequently, the prediction quality will drop. The new defect generation rules have to be fed to the prediction models so that the models comply with this new piece of information.

Table 10 Netbeans: regression model

Feature	Unstand	Standard	*p*
(Constant)	0.6020		0.000
enhancementFixes	0.0003	0.391	0.000
patchFixes	0.0040	0.155	0.000
featureReported	−0.0006	−0.141	0.000
p4-fixes	−0.0001	−0.083	0.035
p5-fixes	0.0024	0.650	0.000
p1-reported	−0.0001	−0.274	0.000
LineOperIRTolLines	0.026	0.057	0.102
author	−0.0007	−0.2490	0.000

Table 11 Performance of the regression models: correlations are significant at $\alpha = 0.01$ level

Project	Pearson	Spearman	MAE	RMSE
Eclipse	0.59	0.308	0.046	0.061
Mozilla	0.57	0.361	0.045	0.057
Netbeans	0.65	0.623	0.041	0.056
Open Office	0.55	0.350	0.066	0.083

Our assumption for the cause of this relationship (between bug fixing effort and the prediction quality) and the different influence in NetBeans is supported by the projects' data: NetBeans has the smallest bug fixing rate per file (3.36) compared to the other three projects (Eclipse: 3.65, Mozilla: 9.77 and Open Office: 9.29) and the Netbean's mean bug fixing rate is significantly different from the other three projects at $\alpha = 0.05$ level. We compute the bug fixing rate per file by dividing the unique number of bugs fixed during the observed periods by the number of files that have at least one bug fixing activity during those periods. Table 12 shows the projects, number of buggy files, mean number of bugs fixed for a file and the standard deviation.

Also, it is worth mentioning that the four regression models use different sets of features for their predictions. One reason could be that the observed projects are completely independent from each other in terms of authors, their workload, their experiences, development environment, etc. Therefore, the set of project features that influence the defect prediction quality varies from one project to another resulting in different regression models. Unfortunately, however, we have no firm theory for why the predictors vary between projects.

Finally, we would like to point out that the somewhat moderate (but significant) correlations reported in Table 11 are no cause for concern. Indeed we embarked on this experiment with the goal of showing that such a prediction of an AUC is possible and produces promising results. We have achieved this goal given the results reported in the table. Obviously, there is room for improvement: a further exploration should consider non-linear regression models.

To conclude, we found that we can predict the AUC of a defect prediction model with a decent accuracy (in terms of mean squared and absolute error). In addition, we found that the feature is consistent in three projects and hence, we further explore this feature in the next section.

5.5 Author Fluctuation and Bug Fixing Activities

The project feature `LineOpeIRTotLines` in the previous experiment encourages us to further investigate the authors' contribution for bug fixing activities. This feature has a negative impact on the AUC of Eclipse, Mozilla and Open office

Table 12 Bug fixing rate per file: mean value of Netbeans project significant at 0.01 level

Project	#of buggy files	Mean	Std. dev.
Eclipse	10,371	3.65	5.7
Mozilla	1,585	9.77	18.8
Netbeans	10,371	3.36	5.7
Open Office	1,832	9.29	10.2

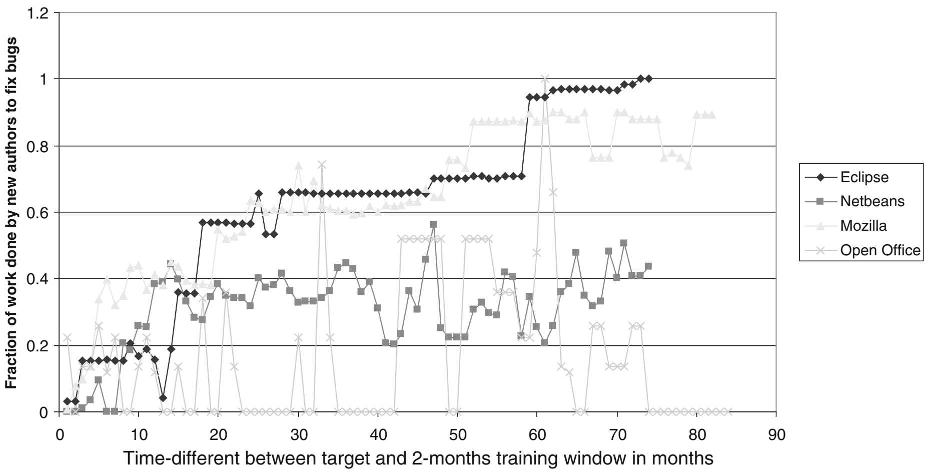


Fig. 10 Work done by new authors to fix bugs

projects. However, this feature does not have any significant effect on the Netbeans model ($p = 10.2\%$). One could, therefore, hypothesize that in Eclipse, Mozilla and Open Office projects most bugs are fixed by authors, who are not active in the training period but in the target period.

To test this proposition we computed the fraction of bug-fixing work done by the authors, who are not in the training period but in the target period. Figure 10 graphs the result for one target period (the last month of the observed period; the others are omitted due to space considerations;[11] but they look similar to this figure), where the x-axis represents the time different between the target period and the two-month training window in months and the y-axis represents the fraction of bug fixing performed by authors, who are not active in the two-months training period, but in the one-month target period.

The figure shows that in Eclipse and Mozilla an increasing proportion of bugs are fixed by those authors, who are not in the training period, and the fraction continuously increases the further we look back into the past. In Open Office the fraction of work done by new authors drastically varies and is probably not meaningful due to a significantly smaller number of transactions (commits) per month. For Netbeans the fraction of work done by authors who are not in the two-months training period to fix bugs is initially very small and never rises above 50% with a mean of 33.2%. Also, the number for Netbeans is relatively constant indicating some stability in its developer base. Hence, authors that are active in the two-months training period fix bugs seems to be increasing the models prediction quality.

The above observations encouraged us to further investigate the relationships between author fluctuation and bug fixing activity in periods of stability versus

[11]You can find a complete set of the figures in the technical report Ekanayake et al. (2011) online. http://www.ifi.uzh.ch/research/publications/technical-reports.html.

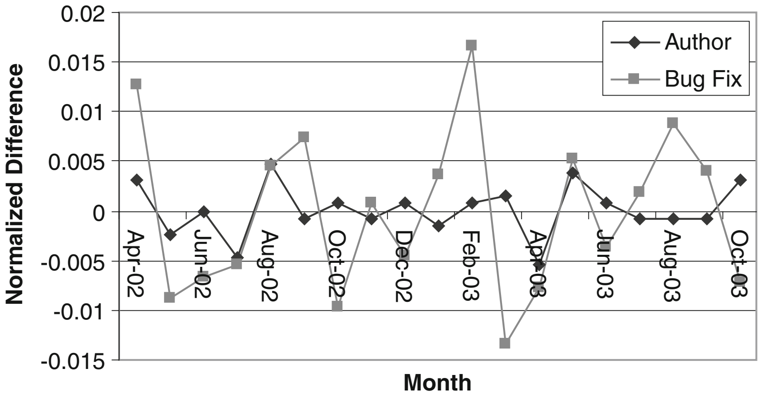


Fig. 11 Eclipse: Tipping starts in July 2003

variability. To that end we identified tipping points from stable to variable periods in each of the projects and graphed the normalized change in number of authors and normalized change in bug fixing activity for the months preceding the onset of the variability and some months into the variability. Consider Eclipse (Fig. 5) as an example: here the investigated months include "stable" months leading up to the tipping month of July 2003 and including the "variable" months until October 2003.

The value for the authors is computed as:

$$authchange_{\text{month}} = \frac{\#auth_{\text{month}} - \#auth_{\text{month}-1}}{\sum_{t \in \text{months}} |\#auth_t - \#auth_{t-1}|}$$

In words: the difference between the number of authors (*#auth*) of the month ($\#auth_{\text{month}}$) and its preceding month ($\#auth_{\text{month}-1}$) normalized by the sum of the

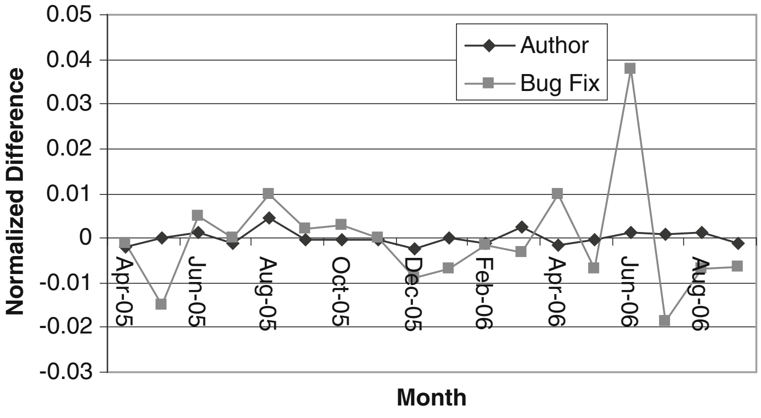


Fig. 12 Netbeans: Tipping starts in April 2006

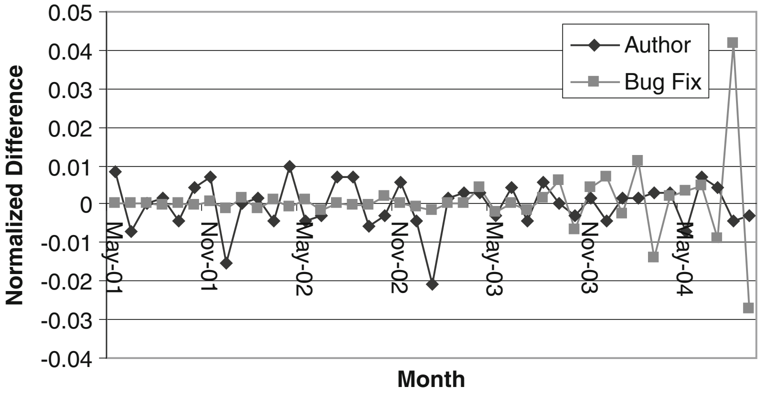


Fig. 13 Open Office: Tipping starts in February 2004

absolute differences of all the months considered in the graph. The value for changes in bug fixes is computed analogously. The rationale for the normalization is to make the figures somewhat comparable across different projects and time-frames.

Figures 11, 12, 13 and 14 show a selection of the resulting figures, which are titled by the "tipping" month.

According to the figures, in most cases we observe a relative drop in the number of authors in the "tipping" month mostly followed by an increase in authors during the drift (decrement in the prediction quality). We also find that in many instances, the relative amount of work done for bug fixing increases in the "tipping" month.

Unfortunately, none of these observations are unique to the tipping periods. Considering Eclipse (Fig. 11), e.g., we find that normalized author differential dips 3 times: in January 03, April 03, and preceding the drop down in prediction quality on July 03. The same can be said for the normalized bug differential. Hence, we cannot argue that these factors can be used exclusively to predict periods when

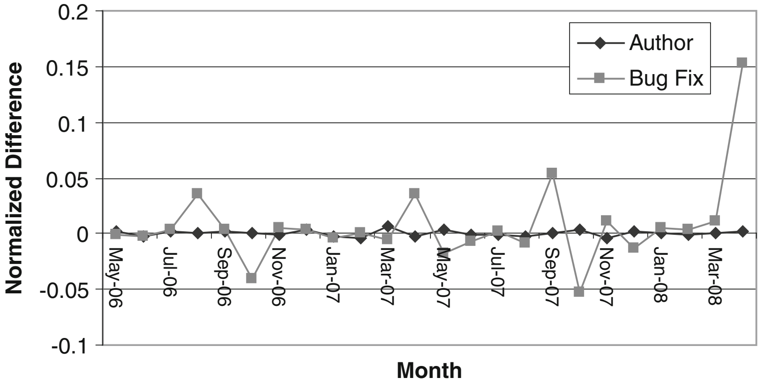


Fig. 14 Open Office: Tipping starts in September 2007

the prediction quality starts declining, but together they can serve as a basis for developing such an early warning indicator.

Summarizing, we observe that increasing the number of authors editing the project has a negative impact on defect prediction quality. We also saw that more work done to fix bugs in relation to the other activities causes a reduction of the defect prediction quality. Further explorations indicated that when authors/developers, who are already present during the learning period and are involving in fixing bugs, helps to increase prediction quality. These finding indicate that it is possible to uncover reasons that influence defect prediction quality.

6 Turning the Insights into Actionable Knowledge

In the previous section we explored the variation in defect prediction quality and early indicators for such a variation. Particularly, in Section 5.4 we found that a prediction model can be learned that predicts the performance of bug prediction models. Our approach essentially devises a prediction model of a prediction model, which we will call 'meta-prediction model' in the following. The main remaining question is, if such a meta-model can be used within a decision procedure for software project managers. In this section, we address this issue and present such a decision procedure that relies on these meta-prediction models.

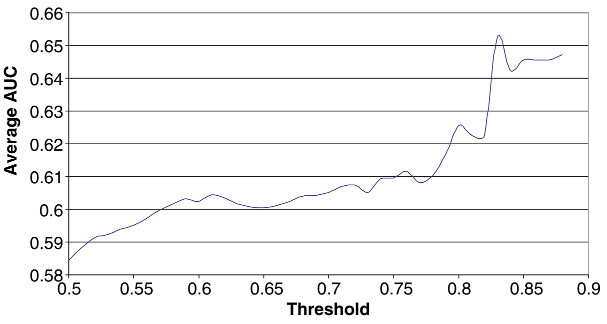
Consider the meta-prediction models learned in Section 5.4 and shown in Tables 7–10. As we argued these meta-prediction models can predict the AUC of the bug prediction model at any given time period using the project features. Assuming that these predictions are good—and we showed in Table 11 that they are at least decent—then it would seem to be natural to use these predictions as a decision measure about the expected quality of bug prediction methods.

Specifically, a software manager hoping to attain a reliable indication for the location and quantity of bugs should only use bug prediction methods when they can be expected to have a certain prediction quality. A good indicator for the expected quality of a bug prediction method that we have is the value generated by the meta-prediction model. Hence, *she should only use the bug prediction method when the meta-prediction model predicts an AUC above a certain threshold*. We have shown in Section 5.2 that the AUC > 0.65 is a "decent" result when compared to BugCache.

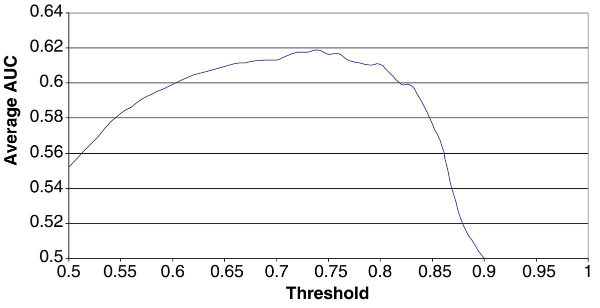
To show that this method works, Fig. 15 graphs the average AUC of all actual predictions gained from the bug prediction models that were predicted to have an AUC above the threshold by the meta-prediction model on the y-axis whilst varying the threshold on the x-axis. Note that whenever the meta-prediction model makes a bad prediction an AUC below the threshold will result.

The figures show that raising the thresholds will eventually lead to better predictions. The figures also show some prediction quality instabilities with a rising threshold, and in three of the four cases a collapse of prediction quality at the very end. Whilst this is disappointing at first it becomes logical when considering that the number of data-points decreases with the rising threshold. In other words: the further right in the figure one looks the less actual predictions are used to compute the average. This has two consequences: First, at the very end only one model is over the threshold and the "average" is really the prediction quality of that model. Second, as the number of data points included in the average decreases, it also becomes

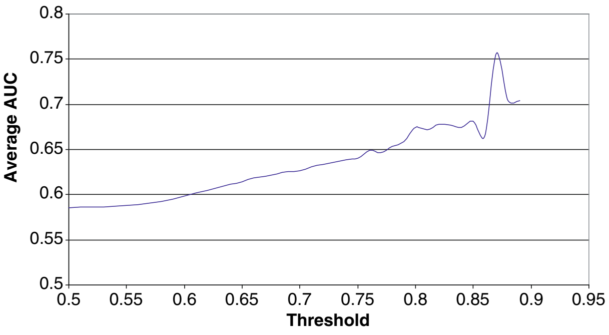
Fig. 15 Graphs estimate the actual AUC based on the predicted AUC by the linear models



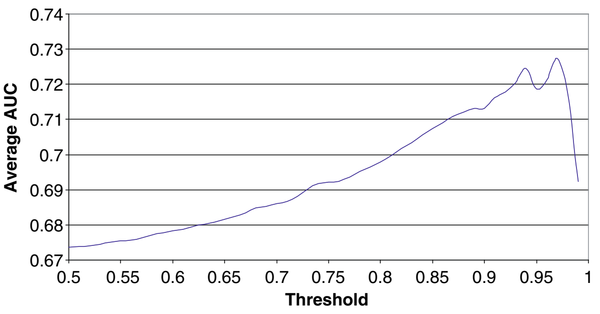
(a) Eclipse



(b) Netbeans



(c) Mozilla



(d) Open Office

increasingly influenced by single misjudgments. Hence, the instability, which initially is quite disappointing becomes understandable.

As a consequence, we can conclude that our approach actually works quite well for all projects. Indeed, choosing thresholds (>0.825 for Eclipse; >0.8 for Mozilla; >0.6 for Open Office) will assure a manager that her model will obtain the minimum required prediction quality (AUC 0.65). However, it does not imply that the model's prediction quality cannot exceed that limit. It can vary in the range of 0.65, which is the minimum in our case, to 1.0. For Netbeans, the threshold is >0.7 and it gives approximately 0.61–0.62 minimum prediction quality. The prediction quality looks rather low. Please note that in Netbeans, we explored 93 subcomponents for this experiment. These subcomponents have been developed under different development environments (different authors, tools, etc.). Therefore, we can expect a large variability in Netbeans data and a consequently lower prediction quality for the model. However, this experiment is not designed for testing the proposition that *the large variability in software data has an impact on defect prediction quality*; this is a venue for future research.

7 Discussion, Conclusions, and Future Work

This paper investigated the notion of periods of stability and variability in data from software projects. Specifically, we were interested in such differing periods with respect to their impact on defect prediction algorithms. Using data from four open source projects we found that the quality of defect prediction approaches indeed varies significantly over time. We, furthermore, found that the quality of the prediction clearly follows periods of stability and variability, indicating that *differing periods are indeed an important factor to consider* when investigating defect prediction. As a consequence, *the benefit of bug prediction in general must be seen as volatile over time and, therefore, should be used with caution*.

We observed that the number of authors editing the project is rising right before, or during periods of instability. This reinforces the well-known software engineering lesson "adding manpower to a late software project makes it even later" (Brooks and Phillips 1995). We also saw a relationship between the changes of the proportion of work done to fix bugs and other activities and the changes in defect prediction quality. Unfortunately, both those correlations were not observed uniformly and can only serve as a start to elicit early warning indicators for changes in stability and, hence, the reduced quality of existing defect prediction models. Furthermore, we found that bug fixing by authors who were active in the learning period helps to improve defect prediction quality. Nonetheless, we found that *we can build a "meta-prediction model" to predict the bug prediction model's quality* with a decent accuracy. This meta-prediction model in turn *can be used as the basis for a decision procedure that allows software managers to decide when to use bug prediction models* and when not. Empirically we found that this decision procedure works well for three out of the four investigated projects. It, therefore, can be seen as a first step to turn insights with regards to stable and volatile phases into an actionable element in software managers' decisions.

During our experimentation we repeatedly asked ourselves if the causes behind the periods of stability and variability lies in *concept drift*. Concept drift is a notion from machine learning that refers to changes in the data generation process. Specifically, Tsymbal (2004) defines is as follows:

> In the real world concepts are often not stable but change with time. . . . Often these changes make the model built on old data inconsistent with the new data, and regular updating of the model is necessary. This problem is known as concept drift, . . . drifts can occur suddenly (abruptly, instantaneously) or gradually.

Our phenomena exhibits strong attributes of concept drift: It shows periods of stability followed by periods of variability (= drift). Indeed, it was this "behavior" that inspired us to draw on some of previous work about prediction under *concept drift* and adapt it to our meta-prediction model (Vorburger and Bernstein 2006). In addition, we found some limited evidence for concept drift: the author fluctuations discussed in Section 5.5, for example, indicate that the influx of new developers is associated with changes in variability. Obviously, these new authors may not be familiar with the norms of the projects and, hence, import new programming habits. This in turn may lead to the introduction of new bugs and changing the concept and generating the periods of variability we observe. Whilst this scenario seems plausible we have no evidence for it. One would have to conduct interviews with developers and extract a multitude of events that may influence a project in order to identify the underlying reasons of the phenomena. While interesting, this is a new study and we and have to leave it open for future work.

Obviously, the generalizability of all our findings is curtailed by the limited number of projects considered. Whilst four projects is a decent size the generalizability of these findings needs to be investigated by looking at additional open and closed source projects. Furthermore, we find that the results are not as 'clean' as one would wish. Indeed, as the Figs. 5–8 illustrate, the triangle shapes indicating periods of stability are sometimes difficult to discern even though we identified and highlighted them using an auto-detection method. We hope that further investigations may uncover the reasons for this seeming 'noise' in the data.

This paper only scratches the surface of changing periods in software projects. Indeed, further investigations into the causes these changes in software projects are needed. In the ideal case it would be possible to identify *the* influential factors that hold for software projects in general. Also, we need to investigate other models for change in addition to our straightforward meta-model. Whatever the outcome of future investigations, we can safely say that the notion of periods of stability and variability seems to have a profound influence on the empirical investigation of software evolution and needs to be taken seriously in any empirical software engineering study—in particular studies about predicting software bugs.

Acknowledgements We would like to thank the anonymous reviewers for their many detailed comments that helped us to significantly improve the paper.

Appendix A: Component List

Tables 13, 14 and 15 list all the investigated components and the number of files that each component consists.

Table 13 Eclipse: Investigated components and number of files

Component	# files	Component	# files
ant_core	36	pde_build	20
ant_ui	294	pde_ui	430
apt_core	161	pluggable_core	10
apt_tests	121	pluggable_tests	5
apt_ui	11	search	126
cknaus	6	text_tests	150
compare	160	ui_home	292
equinox_incubator	5,770	update_core	45
jdt_debug	435	update_home	7
jdt_ui	1,864	update_ui	5

Table 14 Netbeans: Investigated components and number of files

Component	# files	Component	# files
a11y	22	junit	106
accelerators	16	languages	124
ant	221	latex	322
antlr	58	lexer	198
apisupport	319	management	174
archivesupport	31	mdr	324
autoupdate	148	metrics	51
beans	52	mobility	1,412
classclosure	6	monitor	99
classfile	55	nbbuild	111
clazz	26	nbi	278
cnd	1,603	netbrowser	154
codecoverage	39	openide	941
collab	698	performance	343
contrib	1,517	platform	107
corba	513	pluginportal	57
core	1,000	portalpack	253
cpp	40	print	13
cpplite	100	projects	164
db	434	properties	41
debugercore	113	qa	184
debugerjpda	192	refactoring	212
debugertools	21	regsup	110
diff	72	remotefs	21
editor	843	rmi	72
enterprise	4,261	ruby	314
extbrowser	46	schema2beans	94
externaleditor	20	scripting	204
form	472	serverplugins	1,136
freestylebrowser	36	sim	204
graph	287	spellchecker	26
html	93	subversion	151
httpserver	16	tasklist	402
i18n	58	tomcatint	47
ide	183	treefs	47
innertesters	1	ui	65
installer	163	uml	3,757

Table 14 (continued)

Component	# files	Component	# files
j2ee	2,023	utilities	81
j2eeserver	126	vcsgeneric	239
jackpot	89	visualweb	2,410
jasm	76	wasp	183
java	2,085	web	614
javacvs	368	webl	11
javadoc	43	websvc	1,107
jemmy	353	xml	2,102
jemmysupport	23	xtest	209
jndi	73		

Table 15 Mozilla: Investigated components and number of files

Component	# files	Component	# files
accessibl	105	extension	328
browse	22	gf	101
buil	20	int	297
calenda	17	ip	38
camin	6	j	90
cap	6	mai	5
conten	388	mailnew	4
d	62	module	9
director	23	rd	5
do	28	suit	4
docshel	17	widge	5
edito	69	xpf	5
embeddin	242		

Appendix B: Detailed Feature Description

This section describes the features used in the paper as well as explains their computation and rationale.

`revision` We consider a revision as a change made to a file for some reason. The feature `revision` represents the number of changes made to a file during training periods. Both Graves et al. (2000) and Khoshgoftaar et al. (1996) found that past changes are good defect indicators.

`activityRate` This feature measures how often a file has been revised during the training periods and is computed by dividing the number of revisions during the training period by the length of the training period (in months). Hassan and Holt (2005) concluded that a high frequency of changes in a file is a good defect predictor.

`lineAdded`, `lineDeleted` *and* `totalLineOperations` Several studies showed that past changes are good defect predictors (Graves et al. 2000; Khoshgoftaar et al. 1996). Therefore, we further quantify the amount of change done by authors using the features `lineAdded` and `lineDeleted` that describe the number of lines of code added and deleted during training periods. Further, we introduce the total amount of work done for a revision by adding those two features resulting the feature `totalLineOperations`.

`grownPerMonth` This feature provides information about the growth rate of a project or file in the training periods. We compute the amount of grown using the total number of line added and deleted during that time period. Usually, we subtract the total number of line deleted from the total number of line added and then average this value by dividing this number by the length of the training period (in months). Therefore, this number can be ether positive (representing growth) or negative (representing shrinkage). We introduced this feature to address issues that may arise due to too fast change.

`lineOperationRRevision` This feature captures the average size of a revision in terms of number of lines of code added and deleted. We simply add the total numbers of lines of code added and deleted during training periods and divide that amount by the number of revisions during that period.

`chanceRevision` *and* `chanceBug` These two features provide the probability of having a revision or a bug in a file in the future. These features mimic the award winning `BugCache` approach (Kim et al. 2007), which proposes that more recently fixed files are more vulnerable for bugs. We model this probability using the formula $1/2^i$, where i represents how far back (in months) the latest revision or bug occurred from the prediction time period. If the latest revision or bug occurrence is far from the prediction time period, then i is large and the overall probability of having a bug (or revision) in the near future is low.

`blockerFixes`, `criticalFixes`, `majorFixes`, `minorFixes`, `normalFixes` *and* `trivialFixes` These six features report the number of different types of bugs fixed during training periods. The bugs are categorized according to their severity such as blocker, critical, major, minor, normal and trivial. We can find the severity information of fixed bugs from bugzilla database. If a revision has a referenced or linked entry in the bugzilla database and the severity of that entry is marked as one of the above categories then we consider that the revision is for a bug fixing activity. Further, the bug-fixing revision date falls into the training periods then we count as one bug has been fixed in the assigned category. Our intention of introducing these features is to uncover any correlation between the severity and defects.

`enhancementFixes` This feature counts the number of revisions made for enhancements requested during the training period of the models. In the bug categorization process, authors find that some requests are not for fixing bugs, but for enhancements. Hence, we introduce the feature `enhancementFixes` that counts such fixed enhancements.

`blockerReported`, `criticalReported`, `majorReported`, `minorReported`, `normalReported` *and* `trivialReported` These six features provide information about the number of reported bugs in terms of severity. We introduce these features as not all reported bugs during a training period may be fixed within that period. Note that we consider the opening date and the reported dates are same. If an opening date falls into the training period then we count as one bug has been reported in the assigned category.

`enhancementReported` This feature counts number of enhancements reported during training periods. The reported is determined as above.

`p1-fixes`, `p2-fixes`, `p3-fixes`, `p4-fixes` *and* `p5-fixes` Each Bug report is further categorized based on its priority such that the highest and the lowest priority bugs are categorized as P1 and P5 respectively. The other priorities are fallen in between P1 and P5. Theses five features describe the number of priority wise bugs fixed during training periods. Bug fixing dates are determined as in the above cases. If a bug-fixing date falls into the training periods then we count as one bug has been fixed in the assigned category.

`p1-reported`, `p2-reported`, `p3-reported`, `p4-reported` *and* `p5-reported` These five feature provide information about the number of bugs reported with corresponding priority during training periods. The reported dates are determined as in the above.

`lineAddedI`, `lineDeletedI` *and* `totalLineOperationsI` Theses three features provide information about lines of code added, deleted, and total lines of code operated (or changed) to fix bugs during training periods. If a revision has a referenced entry or link in the bugzilla database and the corresponding bug report is not marked as an enhancement but has a severity levels then we consider that revision to be a big fixing activity. Furthermore, the information in the CVS log allows us to extract how many lines of code where added and deleted for that revision supplying the basis for `lineAddedI` and `lineDeletedI`. Adding these two features results in `totalLineOpertaionsI`. These three variables capture how much work (in terms of number of lines of code) is accomlished by the authors to fix bugs.

`lineOperationIRBugFixes` This feature measures the average number of lines of code changed to fix bugs during the training periods. Thus, this features captures the size of the bugs fixed and provides any correlation between the average size of fixed bugs and the defects. We can derive the feature `lineOperationIRBugFixes` by dividing the total number of lines changed to fix bugs by the total number of bugs fixed.

`lineOperationIRTotalLines` This feature describes the work effort by the authors to fix bugs relative to the other work during the training periods. We already computed the total number of lines changed (or operated) to fix bugs and other activities such as enhancements. Hence, we can derive this feature by dividing the total number of lines to fix bugs by the total number of lines changed for any other activity.

`lifeTimeBlocker`, `lifeTimeCritical`, `lifeTimeMajor`, `lifeTimeMinor`, `lifeTimeNormal` *and* `lifeTimeTrivial` These six feature describe about the lifetime of different types of bugs fixed during training periods. Both Bugzilla and CVS databases provide the information about opening and closing dates of the bugs.

Further, Bugzilla provides the severity level of a bugs. Consequently, we can compute the lifetimes of any type of bug by taking the difference between the closing and the opening dates. Note that even when the opening dates lie outside the considered training periods we use them to compute the bug lifetimes.

`hasBug` This is the target variable of some of our models. This variable describes whether any kind of bug (blocking, critical, major, minor, normal, or trivial) has been reported or not in target periods.

Appendix C: Dataset Format

This section describe the format of datasets used in the first experiment.

A dataset contains two parts, labeling and feature computation. The length of the labellings period is usually one month and in this period, we record the number of bugs reported—target variable—for each observed file. The length of the feature computation period—training period—can be extended from one month to the maximum length of the observed period and further, this period starts one month before the labeling period and expands into past. In this period, we compute features listed in Table 2 for each file, which we recorded the number of bugs reported during the labeling period. The description of each feature can be found in the above section. Following is the mathematical notion of the dataset:

Assume that the observed period is d months. $Y_T = \{y_{T,1}, y_{T,2}, ..., y_{T,j}, ..., y_{T,s}\}$ is a vector of dimension s (s is the number of observed files) and $y_{T,j}$ is the number of bugs reported for file j at T, where $1 < T \leq d$. if $X_t = \{f_{t,1}, f_{t,2}, f_{t,i}, ..., f_{t,n}\}$ is a feature vector of dimension n and f$_{t,i}$ is a file feature i computed from the history information at time t, where $n \in N$ and $1 < t \leq d-1$, $t < T$ and $s >>> n$, then constructed dataset is given by $\sum_{t=x}^{T-1} X_t, Y_T$. By changing the x and T variables we can generate different datasets.

References

Ancona D, Chong CL (1996) Entrainment: pace, cycle, and rhythm in organizational behavior. In: Research in organizational behavior, vol 18. JAI Press, Greenwich, pp 251–284

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the Center for Advanced Studies on Collaborative Research (CASCON). ACM, New York, pp 304–318

Bachmann A, Bernstein A (2009) Data retrieval, processing and linking for software process data analysis. Tech. Rep. IFI-2009.0003, University of Zurich, Department of Informatics

Bernstein A, Ekanayake J, Pinzger M (2007) Improving defect prediction using temporal features and non linear models. In: IWPSE '07: ninth international workshop on principles of software evolution, ACM, New York, pp 11–18. doi:10.1145/1294948.1294953

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). ACM, New York, pp 121–130

Brooks FP, Phillips F (1995) The mythical man-month: essays on software engineering. Addison-Wesley, Reading

Diehl S, Gall HC, Hassan AE (2009) Guest editors introduction: special issue on mining software repositories. Empir Software Eng 14(3):257–261

Eaddy M, Zimmermann T, Sherwood KD, Garg V, Murphy GC, Nagappan N, Aho AV (2008) Do crosscutting concerns cause defects? IEEE Trans Softw Eng 34(4):497–515

Ekanayake J, Tappolet J, Gall HC, Bernstein A (2011) Time variance and defect prediction in software projects—additional figures. Tech. Rep. IFI-2011.0004, University of Zurich, Department of Informatics

Fenton NE, Neil M (1999) A critique of software defect prediction models. IEEE Trans Softw Eng 25(5):675–689. doi:10.1109/32.815326

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661. doi:10.1109/32.859533

Hassan AE (2009) Predicting faults using the complexity of code changes. In: ICSE '09: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, Washington, DC, pp 78–88. doi:10.1109/ICSE.2009.5070510

Hassan AE, Holt RC (2005) The top ten list: dynamic fault prediction. In: ICSM '05: Proceedings of the 21st IEEE international conference on software maintenance. IEEE Computer Society, Washington, DC, pp 263–272. doi:10.1109/ICSM.2005.91

Kagdi H, Collard ML, Maletic JI (2007) A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J Softw Maint Evol 19(2):77–131. doi:10.1002/smr.344

Kenmei B, Antoniol G, Di Penta M (2008) Trend analysis and issue prediction in large-scale open source systems. In: Proc 12th European conference on software maintenance and reengineering CSMR 2008. IEEE Computer Society, Los Alamitos, pp 73–82. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4493302

Khoshgoftaar TM, Allen EB, Goel N, Nandi A, McMullan J (1996) Detection of software modules with high debug code churn in a very large legacy system. In: ISSRE '96: Proceedings of the the seventh international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, p 364

Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: ICSE '07: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 489–498. doi:10.1109/ICSE.2007.66

Knab P, Pinzger M, Bernstein A (2006) Predicting defect densities in source code files with decision tree learners. In: MSR '06: Proceedings of the 2006 international workshop on mining software repositories. ACM, New York, pp 119–125. doi:10.1145/1137983.1138012

Ko AJ, Chilana PK (2010) How power users help and hinder open bug reporting. In: CHI '10: Proceedings of the 28th international conference on human factors in computing systems. ACM, Atlanta, pp 1665–1674

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496. doi:10.1109/TSE.2008.35

Li PL, Herbsleb J, Shaw M (2005) Forecasting field defect rates using a combined time-based and metrics-based approach: a case study of openbsd. In: ISSRE '05: Proceedings of the 16th IEEE international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, pp 193–202. doi:10.1109/ISSRE.2005.19

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: ICSM '00: Proceedings of the international conference on software maintenance (ICSM'00). IEEE Computer Society, Washington, DC, p 120

Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: ICSE '05: Proceedings of the 27th international conference on software engineering. ACM, New York, NY, pp 580–586. doi:10.1145/1062455.1062558

Ostrand T, Weyuker E, Bell R (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355

Provost F, Fawcett T (2001) Robust classification for imprecise environments. Mach Learn 42(3):203–231

Quinlan JR (1993) C4.5: programs for machine learning. Morgan Kaufmann, San Mateo

Tsymbal A (2004) The problem of concept drift: definitions and related work. Tech. rep., Department of Computer Science Trinity College

Vorburger P, Bernstein A (2006) Entropy-based concept shift detection. In: ICDM '06: Proceedings of the sixth international conference on data mining. IEEE Computer Society, Washington, DC, pp 1113–1118. doi:10.1109/ICDM.2006.66

Widmer G, Kubat M (1993) Effective learning in dynamic environments by explicit context tracking. In: ECML '93: Proceedings of the European conference on machine learning. Springer, London, pp 227–243

Witten IH, Frank E (2005) Data mining: practical machine learning tools and techniques. Morgan Kaufmann, San Mateo

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: PROMISE '07: Proceedings of the third international workshop on predictor models in software engineering. IEEE Computer Society, Washington, DC, p 9. doi:10.1109/PROMISE.2007.10

Jayalath Ekanayake received his BSc and MSc degrees from the University of Peradeniya, Sri Lanka. He started his PhD at the University of Zurich in December 2006. His main research interest is mining software repositories. Currently, he has been working as a junior lecturer in computer science at the Sabaragamuwa University of Sri Lanka.

Jonas Tappolet is a PhD candidate at the University of Zurich. His research focuses on temporal aspects of graph-based data in the Semantic Web and related fields. Before he started his PhD he studied Business Information Systems at the University of Zurich. He worked in the industry as a software developer for Norwel AG and as a project manager for SAP.

Harald C. Gall received the MSc and PhD (Dr. techn.) degrees in informatics from the Technical University of Vienna, Austria. He is a professor of software engineering in the Department of Informatics at the University of Zurich, Switzerland. Prior to that, he was an associate professor in the Distributed Systems Group at the Technical University of Vienna, Austria.

His research interests include software engineering and software analysis, focusing on software evolution, software quality, software architecture, collaborative software engineering, and service-centric software systems.

He is probably best known for his work on software evolution analysis and mining software archives. Since 1997 he has worked on devising ways in which mining these repositories can help to better understand software development, to devise predictions about quality attributes, and to exploit this knowledge in software analysis tools such as Evolizer.

In 2005, he was the program chair of ESEC-FSE, the joint meeting of the European Software Engineering Conference (ESEC), and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). In 2006 and 2007 he co-chaired MSR, the International Workshop and now Working Conference on Mining Software Repositories, the major forum for software evolution analysis. He will be program co-chair of ICSE 2011, the International Conference on Software Engineering, to be held on the tropical island of Oahu in Hawaii.

Since 2010 he is an Associate Editor of IEEE's Transactions on Software Engineering.

Abraham Bernstein is a Full Professor of informatics at the University of Zurich, Switzerland. His current research focuses on various aspects of the semantic web, knowledge discovery, service discovery/matchmaking, and mobile/pervasive computing. His work is based on both social science (organizational psychology/sociology/economics) and technical (computer science, artificial intelligence) foundations. Mr. Bernstein is a Ph.D. from MIT, where he has played a key role in the development of the Process Handbook (PH), which has been under development at the Center for Coordination Science (CCS).

Prior to joining the University of Zurich Mr. Bernstein was on the faculty at New York University. He also worked for Union Bank of Switzerland, first as a research scientist at the corporate research center for information technology (UBILAB) and then as a project manager for IT-projects, where he worked on a variety of research issues such as HCI for complex tasks, document management, workflow management and data warehousing. Mr. Bernstein also holds a Diploma in Computer Science (comparable to a M.S.) from the Swiss Federal Institute in Zurich (ETH).