Tracking concept drift of software projects using defect prediction quality

Ekanayake, J ; Tappolet, J ; Gall, H C ; Bernstein, A

Abstract: Defect prediction is an important task in the mining of software repositories, but the quality of predictions varies strongly within and across software projects. In this paper we investigate the reasons why the prediction quality is so fluctuating due to the altering nature of the bug (or defect) fixing process. Therefore, we adopt the notion of a concept drift, which denotes that the defect prediction model has become unsuitable as set of influencing features has changed – usually due to a change in the underlying bug generation process (i.e., the concept). We explore four open source projects (Eclipse, OpenOffice, Netbeans and Mozilla) and construct file-level and project-level features for each of them from their respective CVS and Bugzilla repositories. We then use this data to build defect prediction models and visualize the prediction quality along the time axis. These visualizations allow us to identify concept drifts and – as a consequence – phases of stability and instability expressed in the level of defect prediction quality. Further, we identify those project features, which are influencing the defect prediction quality using both a tree induction-algorithm and a linear regression model. Our experiments uncover that software systems are subject to considerable concept drifts in their evolution history. Specifically, we observe that the change in number of authors editing a file and the number of defects fixed by them contribute to a project's concept drift and therefore influence the defect prediction quality. Our findings suggest that project managers using defect prediction models for decision making should be aware of the actual phase of stability or instability due to a potential concept drift.

DOI: https://doi.org/10.1109/MSR.2009.5069480

Tracking Concept Drift of Software Projects Using Defect Prediction Quality

Jayalath Ekanayake*, Jonas Tappolet*, Harald C. Gall+, Abraham Bernstein*
*Dynamic and Distributed Systems Group, +Software Evolution and Architecture Lab
Department of Informatics, University of Zurich
{jayalath, tappolet, gall, bernstein}@ifi.uzh.ch

Abstract

Defect prediction is an important task in the mining of software repositories, but the quality of predictions varies strongly within and across software projects. In this paper we investigate the reasons why the prediction quality is so fluctuating due to the altering nature of the bug (or defect) fixing process. Therefore, we adopt the notion of a concept drift*, which denotes that the defect prediction model has become unsuitable as set of influencing features has changed – usually due to a change in the underlying bug generation process (i.e., the concept). We explore four open source projects (Eclipse, OpenOffice, Netbeans and Mozilla) and construct file-level and project-level features for each of them from their respective* CVS *and* Bugzilla *repositories. We then use this data to build defect prediction models and visualize the prediction quality along the time axis. These visualizations allow us to identify concept drifts and – as a consequence – phases of stability and instability expressed in the level of defect prediction quality. Further, we identify those project features, which are influencing the defect prediction quality using both a tree induction-algorithm and a linear regression model. Our experiments uncover that software systems are subject to considerable concept drifts in their evolution history. Specifically, we observe that the change in number of authors editing a file and the number of defects fixed by them contribute to a project's concept drift and therefore influence the defect prediction quality. Our findings suggest that project managers using defect prediction models for decision making should be aware of the actual phase of stability or instability due to a potential concept drift.*

1. Introduction

In mining software repositories, many different approaches have been developed to predict the number and location of future bugs in source code (*e.g.,* [1], [2], [3], [4], [5]). Such predictions can help a project manager to quantitatively plan and steer the project according to the expected number of bugs and their bug-fixing effort. But bug prediction can also be helpful in a qualitative way whenever the defect location is predicted: testing efforts can then be accomplished with a focus on the predicted bug locations. All of the above mentioned approaches use the history data of a software project to predict defects in the next release. Features (or variables) are extracted from the raw data. These features (from the learning period) are then used together with the goal values (*i.e.,* bug or no bug) to learn a prediction model. To evaluate such a model, it is fed with data from another time period and the predicted values are compared with the observed ones facilitating an accuracy measure.

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 several different points in time. Such selective (insular) analyses make generalizations of the prediction methods difficult: it postulates that the evolution of a project and its data is more or less stable over time.

In our approach we hypothesize that a project passes several alternating phases of stability and instability. Instability can be seen as a sudden change of influencing factors. These factors can be of various kind such as a changing number of developers, the use of a new development tool or even political or economical events (financial crisis, presidential elections) *etc.*

As a consequence, the concept (*i.e.,* the bug generation process) we are trying to learn changes, resulting in a phenomenon called *concept drift* [6]. Obviously, concept drifts can invalidate a learned bug prediction model and lead to less accurate predictions as time progresses. Our goal is to identify and locate concept drifts that affect the accuracy of defect prediction algorithms. For that reason, *our measure for stability and instability of the concept is the quality of the defect prediction*. In a stable phase, the history data is a good predictor for future bugs; analogously, in an unstable phase, the prediction quality will significantly decrease and become unreliable for effort and resource allocation.

Our approach can be summarized as follows. To uncover stable and unstable phases we apply our bug prediction algorithm continuously over time to the data. We provide the algorithm with different temporally sampled feature sets that reflect a changing length of history data available. Hence, for each possible prediction time we evaluate the

Partial support for Jayalath Ekanayake provided by IRQUE fund of the Sabaragamuwa University of Sri Lanka
Partial support for Jonas Tappolet provided by Swiss National Science Foundation award number 200021-112330

quality of bug prediction models learned from every possible (consecutive) period in its past. For example, in month 35 of a given project we use data from the past 34 months to conduct 34 different bug prediction runs each leading its own accuracy value. This method allows us to visualize concept drifts and show that there are indeed phases in a project where a bug prediction is almost useless (with respect to accuracy) and, hence, a project manager should not rely on it. Furthermore, this approach allows us to identify the influencing features that have the potential to serve as early indicators for upcoming concept drifts.

The remainder of the paper is organized as follows: After discussing some related work in Section 2, we describe the experimental setup in Section 3, followed by a discussion of experiments and results. We close with limitations of our study, some possible avenues of future work, and concluding remarks.

2. Related Work

A number of researchers have used the historical data of software projects for different kinds of prediction models. To the best of our knowledge, there is no prior work investigating possible *concept drift* in software projects. However, we here discuss several studies about defect prediction and, as well, related work exploring *concept drift* in different domains.

Bernstein *et al.* [1] used Eclipse's history of product metrics to predict defects. However, the learned model is not evaluated in a temporally continuous way. Instead, only a couple of discontiguous points in time are considered. Since this is our previous work, we use a similar approach to predict the defects in the current experiments as well.

Khoshgoftaar *et al.* [3] used a history of process metrics to predict software reliability and to prove that the number of past modifications of a source file is a significant predictor for its future faults. We also use similar set of features for our work.

Mockus *et al.* [7] studied a large software system to test the hypothesis that evolution data can be used to determine the changes of the software systems and to understand and predict the state of software projects.

Graves *et al.* [2] developed statistical models to determine which features of a module's change history were the best predictors for future faults. They developed a model called *weighted time damp model* which predicted the fault potential by using changes made to the module in the past. We use similar features but we predict the location of the defect.

Hassan *et al.* [4] developed a set of heuristics which highlights the most susceptible subsystems to have a fault. The heuristics are based on the subsystems that were most frequently and most recently fixed. We also compute some of the features that represent the above heuristics for our models. We see most of these features frequently used by our prediction models in stable periods of the projects but not in instable periods.

Nagappan *et al.* [8] presented a method to predict the defect density based on code churn metrics. They concluded that source files with a high activity rate in the past will likely have more defects than source files with a low activity rate. We also added this particular feature set to our prediction models. But none of these features seemed to be of significant influence to a possible explanation of *concept drift*.

Ostrand *et al.* [5] used a regression model to predict the location and number of faults in large industrial software systems. Their predictors for the regression model were based on the code length of the current release and the fault / modification history of the file from previous releases. Although we don't use source code metrics in our study, we extensively use fault / modification histories.

Knab *et al.* [9] predicted defect densities in source code files using decision tree learners. This approach is quite similar to our approach. However, they predicted the number of problems reported. In our models, we predict defect locations. They used both product and process metrics and revealed that process metrics are more significant than product metrics for fault predictions. The model and features, with the exception of the product features, are quite similar to our work. However, they evaluated the model only in very few points in time.

Zimmermann *et al.* [10] proposed a statistical model to predict the location and the number of bugs. They used a logistic regression model to predict the location of bugs and a linear regression model to predict the number of bugs. Further they heavily used product metrics such as *McCabe's Cyclomatic Complexity* as predictors rather than process metrics. In our study, we use decision tree models to predict the location of bugs. Furthermore, we fully rely on process metrics.

Kim *et al.* [11] assumed that faults do not occur in isolation, but rather in bursts of several related faults. They basically considered any location recently changed or recently added together with the known bug is likely to be buggy. We also use some similar metrics such as *chanceBug* in our prediction models.

Brooks *et al.* [12] described in their famous book that adding people to a late project makes it even later. Even our study shows that the number of authors is influencing the stability of the projects.

Tsymbal [6] provided a survey on *concept drift* research in many domains. He argued that in the real world concepts are often not stable but changing over time. He showed typical examples such as weather prediction rules and customer preferences. Furthermore, he mentioned that underlying data distribution may change as well. Also he observed the models built on old data to be inconsistent with new data

and, therefore, regular updating of these models is necessary.

Harries *et al.* [13] explored concept drift in financial time series by using machine learning algorithms. We use a similar approach but with software history data to identify the concept drift.

Widmer *et al.* [14] uncovered from daily experience that the meaning of many concepts heavily depend on implicit context. Changes in that context can cause radical changes in the concept. We argue that the same effect can be observed in software systems.

Kenmei *et al.* [15] showed that the further you go in time the worst will be the prediction, which is also supported by our results.

As a closing remark for this section we like to point out that the idea of *concept drift* per se is not new to the research community. However, software projects have never been subject to such analyses, which is a gap we try to close in this work.

3. Experimental Setup

In this section 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 data for the experiments was extracted from the four open source software projects Eclipse, Netbeans, Mozilla and Open Office. We collected the information provided by CVS and Bugzilla systems for each of the projects. The reason behind selecting these four projects is their long development history (>6 years) that is essential for this kind of analysis to ensure the gathering of multiple developments cycles and their possibly associated drifts. For classification, we use only issues which are marked as defects in the bug database. We understand authorship in terms of the person who brought the changed code into the versioning system rather than the developer who actually wrote the code. This is a necessary simplification since we do not consider the content of files which would shed some light on the real authorship. Table 1 shows an overview of the observation periods and the number of files considered in this work.

Project	First Release	Last Release	#Files
Eclipse	2001-01-31	2007-06-30	9948
Mozilla	2001-01-31	2008-02-29	1896
Netbeans	2001-01-31	2007-06-30	38301
Open Office	2001-01-31	2008-04-30	1847
Total			51,992

Table 1. Considered data sources and time spans.

For all files we exported the history information within the investigated time frames from each project's *Bugzilla* and *CVS* to a *MySQL* database. We then used these data to compute all the features as listed in Table 2. Note that we computed the features on the file level and for each of the available time frames (1, 2, 3, ... months) backwards from the prediction (target) point in time.

Most features' names are self-explanatory but some may need some additional context: The `activityRate` represents how many activities (revisions) took place per month. We include `grownPerMonth`, which describes the evolution of the overall project as a feature (in terms of lines of code). `chanceRevision` and `chanceBug` features describe the probability of having a revision and a bug in future akin to *Bug Cache* [11]. 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 point. If the latest revision or bug occurrence is far from the prediction time point, then i is large and the overall probability of having a bug (or revision) in the near future is low. Hence, these variables model the assumption that files with recent bugs are more likely to have bugs in the future than others (see [4]). `LineOperIRTolLines` represents how many lines were added or deleted to fix a bug in relation to the total number of lines added / deleted. This indicates how much work is currently being done for fixing bugs in relation to other activities (such as adding new features).

3.2. Performance Measures

For most of our experiments we used class probability estimation (CPE) models. In our case the CPE model is a simple decision tree, which computes the probability distribution of a given instance over the two possible classes: `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 accuracy. The problem of the accuracy as a measure is that it does not relate the prediction to the prior probability distribution of the classes. This is especially problematic in heavily skewed distributions such as the one we have (the ratio between defective files and non-defective ones is, depending on the project about 1:20 and approximately remaining this ratio in all samples). Therefore, we used the receiver operating characteristics (ROC) and the area under the ROC curve (AUC), which relate the true-positive rate to the false-positive rate resulting in a measure insensitive to the prior (or distribution) [16]. An AUC close to 1.0 is a good, one close to 0.5 represents a 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

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

Table 2. File features

variables and another variable, called the dependent variable, is modeled by a least squares function, called a linear regression equation. This function is a 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. Experiments: Showing the influence of Concept Drift

In this section we provide empirical evidence regarding the existence of *concept drift* in our four projects. We first show that the defect prediction quality changes over time. Then, in the second experiment we expand on this finding of variability and clearly visualize the periods of stability versus change indicating the existence of concept drift. The third subsection attempts to identify the features relevant for detecting *concept drift*. In other words we try to distill early-warning signs that (i) can be used to caution the usage of results from a bug-prediction model and (ii) might help to unearth the causes for the concept drift.

4.1. Defect prediction quality varies over time

The goal of this experiment is to show that the defect prediction quality varies over time. To that end we employ our features to learn a bug-prediction CPE model for each project. Specifically, we employ Weka's [17] J48 decision tree learner (a re-implementation of C4.5 [18]). To illustrate the large variation of prediction quality over time we trained on data preceding the target month (called the training period), predicted the number of bugs in the target month (or target period), and computed the AUC as a prediction quality measure. For example, if the initial target period is February, 2008, then the initial learning period is January, 2008. We then expanded the training period backwards in time by adding additional data (e.g., from December 2007) from the project's history. Depending on the length of the observation period for a project we could look back up to 74 months for Eclipse and Netbeans, 82 months for Mozilla, and 85 for Open Office. Next, we repeat the procedure by moving the target period one month back and use the preceding periods as training periods. We then visualize the prediction quality (AUC) of each model over time using a heat map (Figure 1 represents Eclipse. We had to omit the other three figures due to space considerations. However, they also exhibit similar characteristics as Eclipse). In the figure the X-axis indicates the target period and the Y-axis the length of the training period (in terms of number of months in past considered). Firstly, it is interesting to observe that in Figure 1, in some periods the model obtains high AUC while others are not. Also, we can see that in some prediction periods, initially the prediction quality is not so impressive but after expanding the learning period up to certain months back the model recovers the prediction quality. However, we can see in some cases that further expansion of learning period from that point could cause degradation of prediction quality. Lastly, it is interesting to observe that once a model has attained a certain accuracy adding additional older information will not destroy it. This indicates that the latest, predictive information is dominating in prediction [4]. The above features

can also be observed in the other three projects.
Summarizing, we clearly show that the defect prediction quality varies over time. This indicates that evaluating a model on one target period or only a few time points is not sufficient. Actually, choosing an optimal target and learning periods can convert a bad prediction model into a usable one and vice versa.

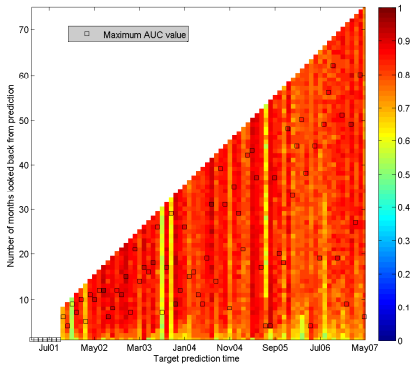


Figure 1. Eclipse: Historical heat-map with the point of highest AUC highlighted

4.2. Finding periods of stability and drift

So far we have seen that the prediction quality clearly varies over time. But are there clear periods of stability and drift (or change)? To clearly differentiate periods of stability and drift we slightly adapted our experiment as follows. Rather than training from the month directly preceding the target period and varying the length of training period we maintain the training period length constant (at 2 months) and move this time window into the past of the project. For example, if the initial target period is February 2008, then the initial training period is December 07 and January 08, followed by a period from November 07 and December 07 *etc.* [4], [1]. We use a 2-month learning window because the typical release cycle of the considered projects is usually 8 to 10 weeks. In addition, our previous work [1] has showed that 2 months of history data attains higher prediction quality. We use Weka's J48 decision tree and we measure the prediction quality using AUC as our first experiment. Again we assign a color for each AUC value and represent it in the heat maps (Figures 2, 3, 4, and 5). 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 more recent of the two months used for building the model. Hence, the higher in the figure we are looking the older is the two-month period compared to the target. Values on the diagonal (bottom left to top right) from each other represent predictions of the model trained on the same period.

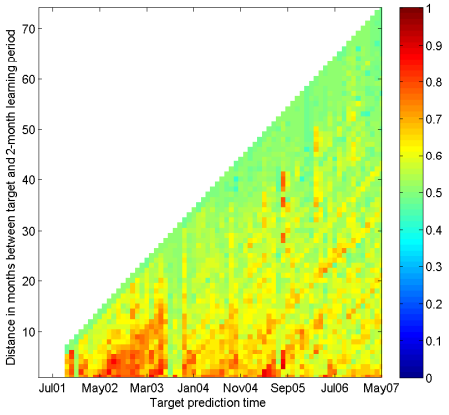


Figure 2. 2-month Heat-map: Eclipse

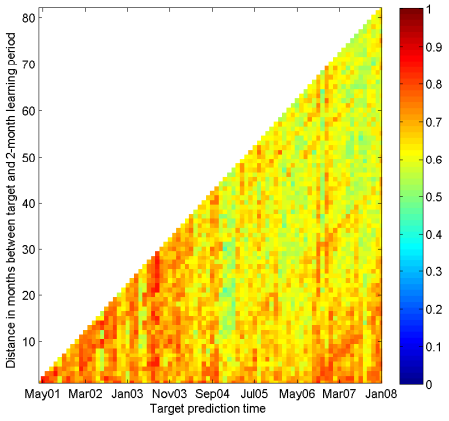


Figure 3. 2-month Heat-map: Mozilla

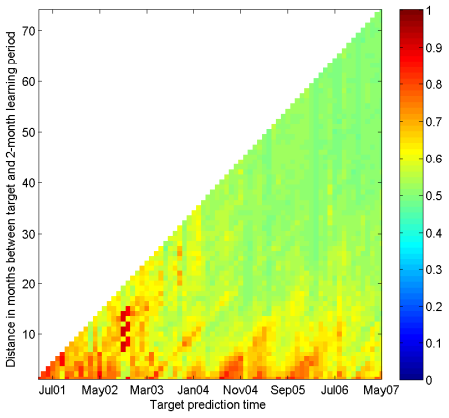


Figure 4. 2-month Heat-map: Netbeans

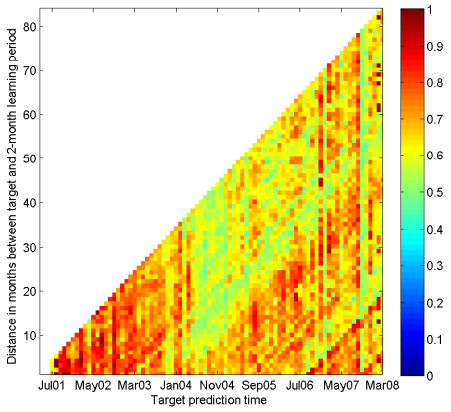
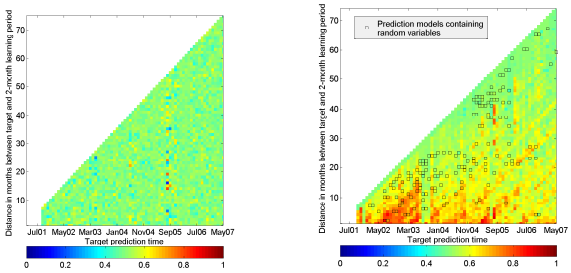


Figure 5. 2-month Heat-map: Open Office

Figure 2 clearly shows different triangle-shapes (red color). One triangle starts from April 2002 and continue to July 2003. In this time period the defect prediction quality is stable at an impressive AUC > 0.8. But suddenly, in August 2003, the defect prediction quality drops to almost random (AUC ≈ 0.5). Hence, the above period is so stable that even models learned on older data (in the summer of 2003 the model trains on data that is older than a year!) have excellent predictive power. This stability results in the triangle shape as the old training (along the upper left boundary/diagonal of the triangle) remains predictive. Also we can observe in all figures that further we go into the past the prediction quality drops down to almost random (≈ 0.5); proves the statement *the further you go in time the worst will be the prediction* [15]. More formally, from April 2002 to July 2003 the concept (*i.e.,* rules underlying the bug generation process and described partially by our features) remains consistent and, hence, the defect prediction quality is stable. Due to the concept drift in August 2003, the defect prediction quality drops down. In January 2004 the project seems to recover some stability and generate another, but slightly less pronounced triangle until November 2004. We can observe the similar effects in NetBeans with much shorter periods of stability and Open Office. In Mozilla this effect seems to be less pronounced maybe as we are really dealing with a set of subprojects. To illustrate that the triangle shapes are not an epiphenomenon of the data or the prediction algorithm, we also graphed the result of a naïve model that simply assumes that the defects of the learning period will be carried over to the target period. As Figure 6(a) clearly shows for Eclipse, most predictions attained in this manner are random (*i.e.,* AUC ≈ 0.5; green in the figure) and do not exhibit the triangle shapes. To prove that the triangles indeed visualize a phenomenon of the underlying data rather than the prediction process itself, we added 10 random variables to our feature set. We then tracked if these variables get picked for a prediction model by the algorithm. Figure 6(b) shows that random variables mostly get included in the model when the AUC is near 0.5 (*i.e.,* close to random). Only a few models inside the triangles contain random variables. Due to space considerations we had to omit the similar figures for the other projects.



(a) Naïve prediction model on Eclipse data

(b) Usage and position of random variables (Eclipse)

Figure 6. Experiments to exclude the possibility of the triangles being an epiphenomenon of the data or the prediction algorithm.

Summarizing, the model clearly exhibits periods of stability and periods of drift. The causes of the drifts – be they observable in our features or not – are not obvious from the graphs and will be investigated in the next subsection. Another interesting observation in the heat-maps is the height of the triangle-shapes. It indicates the length of the stable period. Note, that the height varies both within and between projects. Hence, an universal optimal training period length can not be determined but is highly dependent on the current stable period. Finally, this finding clearly indicates that decision makers in software project should be cautious to base their decisions on a defect prediction model. Whilst they might be useful in periods of stability they should be ignored in periods of drift. In the next experiment we investigate if these periods can be identified from the features we gathered to serve as early warning indicators with regard to the usage of defect prediction models.

4.3. Predicting periods of stability and drift

In the last experiment we show that defect prediction models exhibit periods of stability and drift. But can we uncover features that can be used to predict the kind of period that a software project is in to serve as indicators with regard to the usage of defect prediction models? 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 learning period in the two months before the target period in exactly the same way as in the previous subsection.

Second, since the AUC is a project-level feature of the prediction model we needed project level features to learn the prediction model. Thus, we computed a series of project level features that are listed in Table 7 for each target period. Third, since (i) the AUC prediction model used a 2 months training period and (ii) we are interested in changes between the training and the target 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 value of the target month ($= feature_t - avg_t$). Fourth and last, we build a traditional linear regression model predicting the AUC from these transformed features. The resulting regression models are shown in Tables 3, 4, 5, and 6. Note that 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 less than or equal to 0.05, then those variables significantly contribute to the model, else there is no significant contribution of those variables. The performance of the models is measured in terms of their Pearson correlation, mean absolute error (MAE), and root mean square error (RMSE) as in Table 8. Note that all models have a strong correlation between the predicted and actual values of AUC. Furthermore, the small MAE and RMSE reflect the good performance of our regression models.

Feature	Coefficient	P-value
(Constant)	0.67	0.000
enhancementFixes	0.0002	0.000
enhancementReported	0.0001	0.004
p1-fixes	−0.0013	0.000
p3-fixes	−0.0002	0.000
p5-fixes	−0.043	0.001
p1-reported	0.0015	0.000
p2-reported	0.0001	0.000
p3-reported	−0.0001	0.023
p4-reported	−0.0005	0.000
p5-reported	−0.005	0.000
LineOperationsIRbugFixes	−0.001	0.000
LineOperIRTolLines	−0.1127	0.000
author	−0.0065	0.000

Table 3. Eclipse: Regression Model

Feature	Coefficient	P-value
(Constant)	0.7333	0.000
revision	−0.0001	0.000
bugFixes	0.0001	0.000
enhancementFixes	−0.0012	0.000
enhancementReported	−0.0004	0.000
p1-fixes	0.0004	0.000
p2-fixes	0.0003	0.000
p3-fixes	0.0003	0.000
p4-fixes	0.0012	0.000
p5-fixes	−0.0016	0.000
p3-reported	0.0007	0.000
p4-reported	0.0005	0.001
p5-reported	0.001	0.000
LineOperationsIRbugFixes	0.0011	0.000
LineOperIRTolLines	−0.2478	0.000
author	−0.0007	0.001

Table 4. Mozilla: Regression Model

Feature	Coefficient	P-value
(Constant)	0.67	0.000
bugFixes	−0.0025	0.000
enhancementFixes	−0.0022	0.015
patchFixes	−0.002	0.01
featureFixes	−0.0024	0.000
enhancementReported	−0.0001	0.000
patchReported	0.0001	0.024
p2-fixes	0.0005	0.004
p2-reported	0.0025	0.038
p4-reported	0.0022	0.000
p5-reported	0.0035	0.000
LineOperIRTolLines	−0.0491	0.000
author	−0.0008	0.103

Table 5. Open Office: Regression Model

Feature	Coefficient	P-value
(Constant)	0.602	0.000
enhancementFixes	0.00027	0.000
patchFixes	0.004	0.000
featureReported	−0.0006	0.000
p4-fixes	−0.0001	0.035
p5-fixes	0.0024	0.000
p1-reported	−0.0001	0.000
LineOperIRTolLines	0.026	0.102
author	−0.0007	0.000

Table 6. Netbeans: Regression Model

In all regression models the change in the *number of authors* feature has a negative impact for the AUC. I.e. 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. Hence, the addition of new authors to a project will reduce the applicability of the defect prediction model learned without those authors. Adding new authors could be a cause for concept drift reminiscing the "don't add people to a late project" advice from Fred Brooks' Critical Man Month [12]. The regression coefficients for `author` in all four models are relatively small, but since the AUC moves in the range of $0.5 - 1.0$ they contribute about 1% to the model providing at least a qualitative indication.

Another interesting feature of the models is `LineOpeIRTotL`: number of lines added / removed to fix bugs relative to total number of lines operated. This feature reflects the fraction of work performed to fix bugs relative to total work done. In all of these models this factor has high impact on the models, since it has the highest coefficient. In Eclipse, Mozilla, and Open Office, this factor contributes negatively to the model, while in Netbeans it contributes positively. The higher this value, the more bugs are fixed in the next version, the lower the more new

Name	Description
revision	Number of revisions
grownPerMonth	Project grown per month
totalLineOperations	Total number of line added and deleted
bugFixes	Total number of bugs fixed in every type
bugReported	Total number of bugs reported in every type
enhancementFixes	Number of enhancement requests fixed
enhancementReported	Number of enhancement requests Reported
p1-fixes	# of priority one bugs fixed
p2-fixes	# of priority two bugs fixed
p3-fixes	# of priority three bugs fixed
p4-fixes	# of priority four bugs fixed
p5-fixes	# of priority five bugs fixed
p1-reported	# of priority one bugs reported
p2-reported	# of priority two bugs reported
p3-reported	# of priority three bugs reported
p4-reported	# of priority four bugs reported
p5-reported	# of priority five bugs reported
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
totalLineOperationsI	Total lines operated to fix bugs
lineOperationIRbugFixes	Average number of lines operated to fix a bug
lineOperationIRTotalLines	Number of lines operated to fix bugs relative to total line operated
lifeTimeIssues	Average lifetime of all types bugs
lifeTimeEnhancements	Average lifetime of enhancement type bugs
authors	Total number of authors
workload	Average work done by an author
AUC	Area under ROC curve (Target)

Table 7. Project features

Project	pearson correlation	MAE	RMSE
Eclipse	0.59	0.046	0.061
Mozilla	0.57	0.045	0.057
Netbeans	0.65	0.041	0.056
Open Office	0.55	0.066	0.083

Table 8. Performance of the regression models

features are introduced. Hence, if the coefficient is negative as in Eclipse, Mozilla, and Open Office, then more new features are added than bugs fixed presumably leading to some stability with regards to bugs. Further we can support for the above statement since the Netbeans project has the smallest bug fixing rate per file (0.32) compared to the other three projects (Eclipse: 0.43, Mozilla: 3.36 and Open Office: 0.94).

One important issue to note is that whilst `LineOpeIRTotL` contributes strongly to the Netbeans model, it does not do so significantly ($p = 10.2\%$). One could, therefore, hypothesize that in Netbeans, in contrast to the other projects, most bugs are fixed by experienced authors whose behavior is well captured by the model. To test this proposition we computed the fraction of work done by the authors, who are not in the learning period but in target period, to fix bugs. Figure 7 graphs the result for one target period (the others are omitted due to space considerations), where the X-axis represents time into the past from the target period and the Y-axis represents the fraction of bug fixing performed by new authors. The figure clearly shows that in Eclipse and Mozilla most of bugs are fixed by those authors, who are not in the learning period and the fraction continuously increases the further we look back into 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 new authors to fix bugs is initially very small and does never rise above about 50% with a mean well below 40%. Also, the number for Netbeans is relatively constant indicating some stability in its developer base. Hence, mostly experienced authors seem to be fixing bugs increasing the models prediction quality as those authors behavior is already known in the learning period.

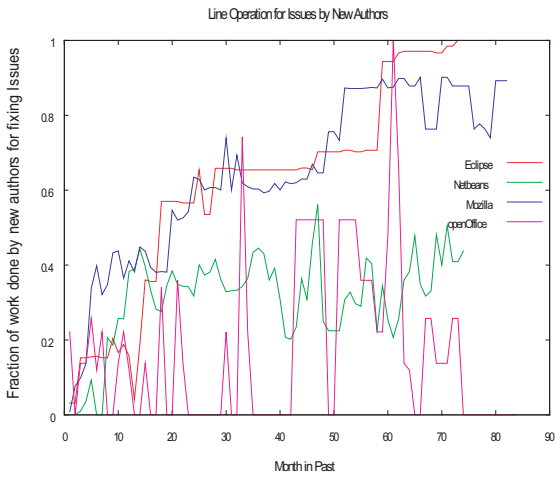


Figure 7. Work done by new authors to fix bugs

Note that the feature `enhancementFixes` is occupied by all four regression models. However, this feature is not consistent since in Open Office and Mozilla it contributes negatively while in Netbeans and Eclipse it is positive. Therefore, it is difficult to figure out the behavior of this feature in the context of software engineering.

Summarizing, we observed that rising the number of authors editing the project could cause the drop of the defect prediction quality. We also saw that more work done to fix bugs relative to the other activities as well causes a reduction of the defect prediction quality. Therefore, the behavior of these two features could be considered as an early warning signal for *concept drift*.

Exploring author fluctuations: The above observations encouraged us to further investigate the relationships between

author fluctuation, bug fixing activity, and stable versus drift periods. To that end we identified tipping points from stable to drift 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 drift and some months into the drift. Consider Figure 2 as an example, the "stable" months leading up to the tipping month of August 2003 and including the "drifting" month of October 2003. The value for the authors are computed as shown below.

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

In words, the difference between the number of authors ($\#auth$) of the month and its preceding month normalized by the sum of the 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 8, 9, 10, 11, and 12 show a selection of the resulting figures, which are titled by the "tipping" month.

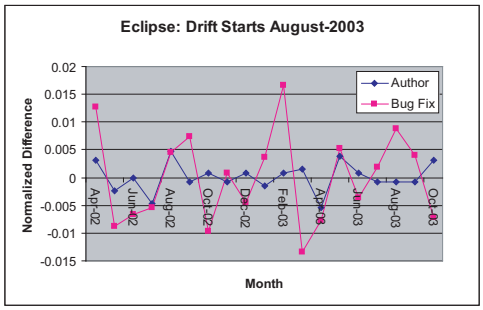


Figure 8. Eclipse: Drift starts in August 2003

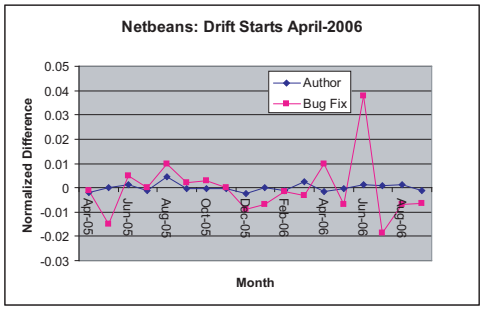


Figure 9. Netbeans: Drift starts in April 2006

All five figures show a relative drop in authors before or in the "tipping" month mostly followed by an increase in authors during the drift. We also find that in most cases, the relative amount of work done for bug fixing increases massively in the first month of the drift. Unfortunately,

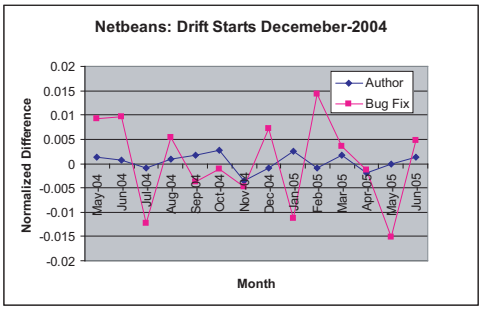


Figure 10. Netbeans: Drift starts december 2004

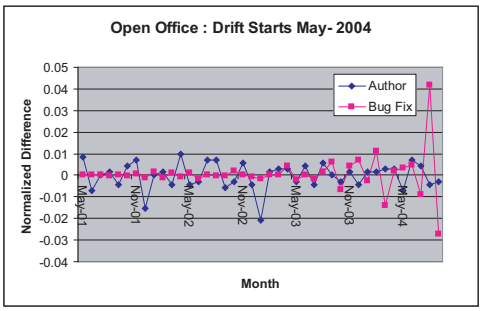


Figure 11. Open Office: Drift starts in May 2004

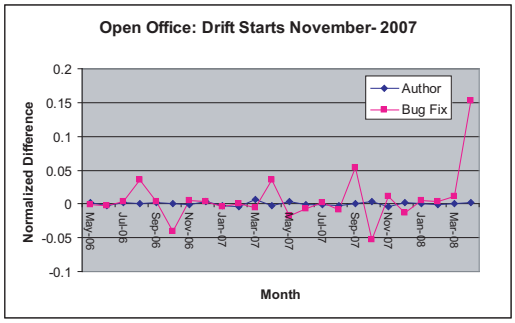


Figure 12. Open Office: Drift starts in November 2007

neither of these observations is unique to the tipping periods. Considering Eclipse (Figure 8), *e.g.,* we find that normalized author differential tips 3 times: in January 03, April 03, and preceding the drift in 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 of drift, but together they can serve as a basis for developing such an early warning indicator.

Summarizing, this third experiment shows that the prediction of drift periods seems to be possible and pursuing early warning indicators for drifts seems to be a promising endeavor. In addition, the results highlight that author / developer fluctuations as well as changes in the amount of work expended to fix bugs in relation to adding new features seem to *correlate* with changes in prediction quality.

From a software engineering standpoint these correlations can definitely be explained and would uphold some time-honored principles.

5. Conclusions and Future Work

This paper investigated the notion of *concept drift* in data from software projects. We were specifically interested in drifts of the concept "bug generation process" as it would impact 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 drift, indicating that concept drift *is 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 cautiously*.

In a further experiment we attempted to uncover the underlying causes of *concept drift* in a software project. We observed that number of authors editing the project is rising right before, or during a *concept drift*. This reinforces the well-known software engineering rule "adding manpower to a late software project makes it later"[12]. We also saw a relationship between the changes of the proportion of work done to fix bugs and other activities and the defect prediction quality. Unfortunately, both those correlations were not observed uniformly in connection with *concept drift* and can only serve as a start to elicit early warning indicators for *concept drift* and, hence, the reduced quality of existing defect prediction models. We plan to further investigate the question about the causes of *concept drift* in software projects. In the ideal case it would be possible to identify *the* influential factors that hold for software projects in general. Whatever the outcome of our future investigations, we can safely say that the notion of *concept drift* seems to have a profound influence in the empirical investigation of software evolution.

References

[1] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Proceedings of the International Workshop on Principles of Software Evolution*, 2007.

[2] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[3] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the 7th International Symposium on Software Reliability Engineering*, 1996.

[4] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in *Proceedings of the 21st International Conference on Software Maintenance*, 2005.

[5] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[6] A. Tsymbal, "The problem of concept drift: Definitions and related work," Department of Computer Science Trinity College, Tech. Rep., 2004.

[7] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*, 2000.

[8] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005.

[9] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 international workshop on mining software repositories*. ACM, 2006.

[10] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007.

[11] S. Kim, T. Zimmermann, E. J. W. Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[12] F. P. Brooks and F. Phillips, *The mythical man-month: essays on software engineering*. Addison-Wesley Reading, MA, 1995.

[13] M. Harries and K. Horn, "Detecting concept drift in financial time series prediction using symbolic machine learning," in *Proceedings of the 8th Austrailian Joint Conference on Artificial Intelligence*. World Scientific Publishing, 1995.

[14] G. Widmer and M. Kubat, "Effective learning in dynamic environments by explicit context tracking," in *Proceedings of the European Conference on Machine Learning*, 1993.

[15] B. Kenmei, G. Antoniol, and M. D. Penta, "Trend analysis and issue prediction in Large-Scale open source systems," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008.

[16] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine Learning*, vol. 42, no. 3, 2001.

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

[18] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.