Improving Defect Prediction Using Temporal Features and Non Linear Models

Abraham Bernstein
Department of Informatics
University of Zurich
Switzerland
bernstein@ifi.unizh.ch

Jayalath Ekanayake
Department of Informatics
University of Zurich
Switzerland
jayalath@ifi.unizh.ch

Martin Pinzger
Department of Informatics
University of Zurich
Switzerland
pinzger@ifi.unizh.ch

ABSTRACT

Predicting the defects in the next release of a large software system is a very valuable asset for the project manger to plan her resources. In this paper we argue that *temporal features (or aspects) of the data are central to prediction performance.* We also argue that *the use of non-linear models, as opposed to traditional regression, is necessary to uncover some of the hidden interrelationships between the features and the defects* and maintain the accuracy of the prediction in some cases.

Using data obtained from the CVS and Bugzilla repositories of the Eclipse project, we extract a number of temporal features, such as the number of revisions and number of reported issues within the last three months. We then use these data to predict both the location of defects (i.e., the classes in which defects will occur) as well as the number of reported bugs in the next month of the project. To that end we use standard tree-based induction algorithms in comparison with the traditional regression.

Our non-linear models uncover the hidden relationships between features and defects, and present them in easy to understand form. Results also show that using the temporal features our prediction model can predict whether a source file will have a defect with an accuracy of 99% (area under ROC curve 0.9251) and the number of defects with a mean absolute error of 0.019 (Spearman's correlation of 0.96).

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement; D.2.8 [**Software Engineering**]: Metrics

Keywords

Mining Software Repository, Defect Prediction, Decision Tree Learner

1. INTRODUCTION

One of the central questions in software engineering is how to write bug-free software. Given that it is virtually impossible to do so researchers are striving to develop approaches for predicting the location, number, and severity of future/hidden bugs. Such predictions can be used by software development managers to (among other things): (1) identify the most critical parts of a system that should be improved by respective restructuring, (2) try to limit the gravity of their impact by, e.g., "avoiding" the use of these parts, and/or (3) to plan testing efforts (parts with most defects should be tested most frequently).

Several approaches have been developed to predict future faults using historical data [2, 3, 5, 11], however many of them have not been evaluated or are not applicable to large software systems. Addressing these two issues our long term objective is to develop an easy-to-use tool for predicting future defects in source files akin to the Hatari tool described in [16]. In this paper we present a number of experiments to investigate the significance of temporal features and the applicability of non-linear models for predicting whether a source file will have a defect and the number of defects. A well performing prediction model is key for our tool.

For our experiments we employ six plugins of the Eclipse project. For each plugin we obtain historical data from the issue tracking system Bugzilla[1] and the version control system CVS.[2] Based on these data we compute a number of features of the actual source code, past defects (bugs), and modifications. In our experiments we test different feature sets to find out the most significant one. For the prediction of the location of defects we use a decision tree learner as has been also used in one of our previous experiments presented in [7]. For the prediction of the number of defects in source files we use a regression tree learner (in addition to traditional regression). Referring to the previous application examples we used the results of our predictions to identify the Eclipse plugins (out of the six) that should be refactored and tested with care.

The results of experiments show that the use of *temporal features* significantly improves the performance of prediction models (both, location and number of defects). Furthermore, we show how the use of *non-linear models* helps to uncover some of the non-linear relationships between features as well as between the feature and the target variables (i.e., defect location and number of defects) improving prediction performance. Our model exhibits excellent results: we are able to predict the defect location with an

IWPSE'07, September 3-4, 2007, Dubrovnik, Croatia.

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

accuracy of 99% (given a base rate of 96.3% and 3.6%) resulting in a distribution independent area under the ROC curve of 0.9251 (see [12]). The number of defects prediction model also exhibits an excellent prediction resulting in a Spearman's correlation of 0.96 and a mean absolute error of 0.0194. Furthermore, we observed that (1) the best predictor for defects in a source file was the past existence thereof [3], (2) some features collected were actually detrimental to the overall performance, and (3) the process measures based on the change history were better predictors than the traditionally used code-metrics, which is supported by several recent studies [2, 7].

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

2. RELATED WORK

The historical data of software systems is a valuable asset used for research ranging from software design to software development, software maintenance, software understanding, and many more. A number of researches use the historical data of software projects for their research in the above fields. We list here few of the studies similar to our study.

Khoshgoftaar et al.[5] 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.
Mockus et al.[9] 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. Our approach also supports this idea.
Graves et al. [2] developed statistical models to find 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 using changes made to the module in the past. We use similar features but employ non-linear models.
Hassan et al. [3] 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. Our approach provides some matrices to represent the above heuristics.
Nagappan et al.[10] presented a method to predict defect density based on code churn metrics. They found out that source files with a high activity rate in the past will likely have more defects than source files with a low activity rate. They pointed out that the relative measures are better predictors for defects than the absolute measures. In our experiment, all the measures are relative and moreover we used machine learning techniques in addition to the linear regression model to predict the number of defects.
Ostrand et al. [11] used a regression model to predict the location and number of faults in large industrial software systems. The predictors for the regression model were based on the code length of the current release, and fault/modification history of the file from previous releases. Our study also supports the significance of the modification reports and the number of reported problems for defect prediction but does not support the significance of the code length.
Knab et al. [7] presented a method to predict defect densities in source code files using decision tree learners. This approach is quite smiler to our approach. However they predicted only the number of problems reported. In our models, we predict both the number of problems and the locations. They used both product and process measures for the defect prediction and revealed that process measures are more significant indicators for fault prediction than product measures, which is also supported by our findings.
Askari et al. [1] presented three probabilistic models to predict the number of defects of source files. They used an information theoretic approach and pointed out that the predictive rate of modification in a file is incremented by any modification to that file and decay exponentially. In our study we also use past modification reports as an indicator of defects.
Finally, Zimmermann et al. [18] proposed a statistical model to predict the location and the number of bugs. They used logistic regression model to predict the location of bugs and the linear regression model to predict the number of bugs. Further they heavily used product metrics such as McCabe cyclomatic complexity as predictors than process metrics. In this study, we use non-linear decision tree models to predict the location and the number of bugs and show that they are superior to linear ones. Furthermore, we heavily rely on process metrics than product metrics.

Our approach takes guidance from these approaches. It seems to be the first to combine the use of temporal features with non-linear models.

3. EXPERIMENTAL SETUP

In this section we succinctly introduce the overall experimental setup. We discuss the data used and measures used to judge the quality of the results.

3.1 The Data - CVS and Bugzilla for Eclipse

The data for the experiment was extracted from six plugins of the Eclipse open source project: `updateui`, `updatecore`, `search`, `pdeui`, `pdebuild`, and `compare`. For each plugin we considered the CVS and Bugzilla data from the first releases up to the last one released in January 2007 as provided by the MSR Mining Challenge 2007.[3] Table 1 lists the release dates and the number of files (taken from the last release).

Plugin	First Release	Last Release	#Files
updateui	Jan 03, 2001	Jan 18, 2007	757
updatecore	Jan 03, 2001	Jan 18, 2007	459
search	May 02, 2001	Jan 30, 2007	540
pdeui	Mar 26, 2001	Jan 30, 2007	1621
pdebuild	Dec 11, 2001	Jan 12, 2007	198
compare	May 02, 2001	Jan 30, 2007	315
Total			3890

Table 1: Eclipse plugins considered

Of the 3890 files we omitted 59 as they did not have a sufficient number of revisions to provide temporal information for our experiment. Other examples for exclusion

[3] http://msr.uwaterloo.ca/msr2007/challenge/

#	Name	Description
1	LOC	Number of lines of codes
2	LineAddedIRLAdd	Number of lines added to fix a bug relative to total number of lines added
3	LineDeletedIRLDel	Number of lines deleted to fix a bug relative to total number of line deleted
4	AlterType	Amount of modification done relative to LOC
5	AgeMonths	Age of a file in months
6	RevisionAge	Number of revisions relative to the age of a file
7	DefectReleases	Number of releases of a files with defects relative to total number of releases
8	Revision1Month	Number of revisions of a file from Dec 1 to 31 of 2006
9	DefectAppearance1Month	Number of releases of a file with defects from Dec 1 to 31 of 2006
10	ReportedI1Month	Number of reported problems of a file from Dec 1 to 31 of 2006
11	Revision2Months	Number of revisions of a file from Nov 1 to Dec 31 of 2006
12	DefectAppearance2Months	Number of releases of a file with defects from Nov 1 to Dec 31 of 2006
13	ReportedI2Months	Number of reported problems of a file from Nov 1 to Dec 31 of 2006
14	Revision3Months	Number of revisions of a file from Oct 1 to Dec 31 of 2006
15	DefectAppearance3Months	Number of releases of a file with defects from Oct 1 to Dec 31 of 2006
16	ReportedI3Month	Number of reported problems of a file from Oct 1 to Dec 31 of 2006
17	Revision5Months	Number of revisions of a file from Aug 1 to Dec 31 of 2006
18	DefectAppearance5Months	Number of releases of a file with defects from Aug 1 to Dec 31 of 2006
19	ReportedI5Month	Number of reported problems from Aug 1 to Dec 31 of 2006
20	ReportedIssues	Total number of reported problems
21	Releases	Total number of releases
22	RevisionAuthor	Number of revisions per author

Table 2: The features (or measures) used in our experiment

were files with modification reports that do not contain lines added/deleted information or with a wrong or unavailable release date. We exported all the information into the evolution ontology format EvoOnt data [6], which integrates the code, release, and bug information in a single knowledge base. For each of the investigated 3831 source files we used the information in the EvoOnt knowledge base to compute the number of lines of code (LOC) code and several process features (or measures) as listed in Table 2. Features 8–19 contain temporal/historical information about the project. Essentially, they consider different sizes of windows (1, 2, 3, and 5 months) backwards from the December 2006 releases. If a defect is not fixed in one release and transferred to later releases, then we count them in all releases where they occur.

Since lines of codes are added/deleted both when fixing a bug and when adding new features they need to be separated. Features LineAddedIRLAdd and LineDeletedIRLDel represent the number of lines added/deleted to fix a bug relative to total number of lines added/deleted.

Feature 4, AlterType, classifies each modification into large, medium, and small according its size relative to the lines of code modified in the source files. If the sum of lines added and deleted is more than double of the code length then AlterType of this modification is large. If the modification relative to the code length is between 1 and 2 then Altertype is medium. If the size of the change is below 1 than AlterType is small. This reflects the way modifications are handled by CVS, which stores for a modified line 1 line deleted and 1 line added.

3.2 Experimental Procedure

All experiments were carried out using the Weka data mining toolkit [17]. To test the quality for our prediction models we computed the features shown in Table 2 once for releases until December 31 2006 for learning/inducing the model – the *training set* – and once for the period until the end of January as a *test set*.

Since choosing a good feature set for the prediction model is imperative for a good prediction performance we used a number of *wrapper-based feature selection* methods such as sequential forward selection [8]. These methods compare the prediction performance of different subsets of the features within the training set to find the best performing subset.

The best performing subset of features was then used to induce the prediction model, which was then tested on the test set. This procedure ensures that only information available on December 31, 2006 was used to predict the location of defects or the number of bugs in January of 2007.

3.3 Performance Measures

For the location prediction experiment we learned a class probability estimation model (CPE), which computes the probability distribution over the two possible classes: hasBug and hasNoBug. Since CPE's are usually used to predict classes we picked the class with the higher probability and computed the confusion matrix of the model, which can (partially) be summarized with accuracy of the model's classification. The problem of the accuracy as a measure is that it does not relate the prediction the prior probability of the classes. This is especially problematic in heavily skewed distributions such as the one we have. Therefore, we also used the receiver operating characteristics (ROC) and the area under the ROC curve, which relate the true-positive rate to the false-positive rate resulting in a measure uninfluenced of the prior (or distribution) [12, 17].

Given the skewed distribution the traditional Pearson correlation is inappropriate. For the regression experiment we, therefore, report Spearman's Rank correlation (ρ), root mean squared error (RMSE), and mean absolute error (MAE).

4. EXPERIMENTS

In our experiments we investigate the suitability of our approach for two tasks. First, we looked if the features selected are sufficient to predict the files that will have *defects in future versions*. Second, we explore how well our approach predicts the *number of bugs* per each file.

4.1 Defect Location Prediction

The goal of this experiment is to predict the locations of defects of source code files. To that end we learn a model using the training data that predicts the probability of defect occurrence for any given file from the test set. We used Weka's J48 decision tree learner (a re-implementation of C4.5 [13]). To test our proposition—that temporal features would improve the prediction quality—we learned the model with different base-sets of features either using no temporal data whatsoever (i.e., excluding features 8-19 of Table 2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months (i.e., choosing a selection of features 8-19 representing the window size under investigation).

Table 3 summarizes the results of these experiments. It shows the list of features chosen by the feature selection method, the accuracy, and the area under the ROC curve of the prediction for each of the learned models. It is interesting to observe that the only feature chosen for all models is the `LineAddedIRLAdd` (Feature 2), which relates the numbers of lines added due to bug fixing to the number of lines added due to adding new features. Even though this feature is chosen by all models it does not seem to play a pivotal role in the models, as it does not show in none of the trees' first two levels. Another interesting observation is the dominance of the temporal features (numbers 8-19): not only do they get chosen whenever possible, they also show up at the root of the tree (see column 3) whenever available.

When looking at the target performance measures accuracy and area under the ROC curve (AUC) we clearly see the dominance of the prediction that can take advantage of temporal features compared to the one that cannot. In terms of accuracy we can clearly see that temporal information boosts the performance, but that more recent temporal data is more useful than older one. We can, hence, hypothesize that modules with bugs are likely to have bugs in later versions, but that over longer periods of time those bugs could be fixed. In other words: Bugs are likely to survive some versions, but are fixed after some.

One might argue that the difference between 96.58% (no temporal features) and 99.16% (significant features) in accuracy is not significant enough to warrant the computation of the temporal measures. Note, however, that the sole use of accuracies is misleading since they are heavily dependent on the prior distribution of the data. In our case, where the class distribution is highly skewed (we have 140 buggy classes versus 3691 non-buggy ones), it is simple to attain a high accuracy: "just" assigning "non-buggy" to every file (the default strategy) one gets an accuracy of 96.35% ($= \frac{3691}{3691+140}$) according to the confusion matrix for the best model (including significant features) shown in Table 4. Hence, the use of accuracy as a measure for the quality of the prediction is misleading. We, therefore, computed the receiver operating characteristics (ROC) for each of the methods and the area under the ROC-curve (AUC), which both provide a prior-independent approach for comparing the quality of a predictor [12].

Figure 1 graphs the ROC curves for all the chosen methods. The x-axis shows the false-positive rate and the y-axis the true positive rate. Note, that a random bug assignment is also shown as a line form the origin $(0, 0)$ to $(1, 1)$ and that the ideal ROC curve would be going from the origin straight up to $(0, 1)$ and then to $(1, 1)$. The Figure clearly shows that all prediction methods provide a significant lift in predictive quality over the random assignment. But the methods have very interesting differences in terms of quality. Since one method dominates another when its ROC-curve is closer towards the upper left corner, we can see how the non-temporal prediction model is dominated along almost the whole frontier by the temporal models. The figure also shows how the method using significant features dominates the other methods along almost the whole frontier whilst employing fewer features (see Table 3). Lastly note, that the dominance of the ROC-curve is reflected by a larger area under the ROC curve (AUC) as listed in Table 3.

	predicted buggy	predicted bug free
has bugs	117	23
has no bugs	9	3682

Table 4: Confusion Matrix for the significant features model

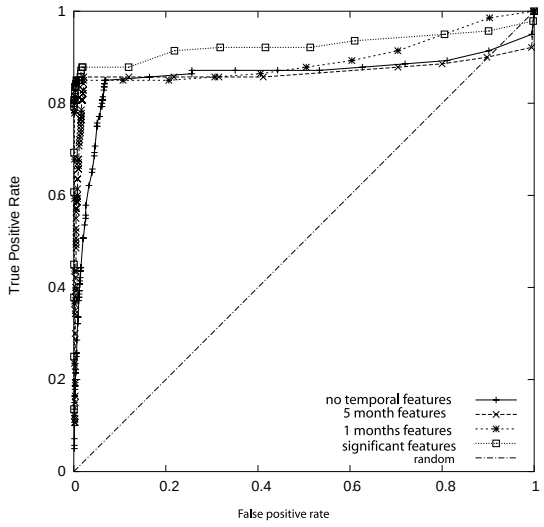

Figure 1: ROC-curves of defect prediction methods.

To further improve our understanding of the structure of the prediction methods, we succinctly compare the two top levels of the prediction trees for the non-temporal, the 1-month, and the significant feature model. As the top levels of the trees depicted in Figure 2 show even the model without temporal features (a) heavily relies on the quasi temporal feature `DefectReleases`, which computes the fraction of past releases with bugs. The next most important non-temporal feature seems to be `LineDeletedIRLDel` signifying the importance to distinguish between changes due to bug fixing versus changes due to the addition of new features. Not that this seems to be a very important distinction, as the related `LineAddedIRLAdd` feature is the most important non-temporal feature in the tree (b).

Name of method	Features chosen	Root Node of Tree	Accuracy	Area under ROC curve
no temporal features	2,3,4,6,7,20,21,22	`DefectReleases`	96.5805%	0.8611
1-month features	2,8,10,20,22	`ReportedI1Month`	99.1125%	0.8948
2-months features	2,4,11,12,13,20,22	`ReportedI2Month`	98.8776 %	0.8933
3–months features	2,3,4,6,7,14,15,16,20,21,22	`DefectAppearance3Months`	98.6427%	0.9039
5-months features	2,3,4,5,6,7,17,19,20,21,22	`ReportedI5Months`	97.7813%	0.8663
significant features	2,3,8,9,11,16,19	`DefectAppearance1Month`	99.1647%	0.9251

Table 3: Results of different models for defect location prediction (Accuracy of default strategy 96.35%)

```
DefectReleases <= 5.263158: NO (3423.0/11.0)
DefectReleases > 5.263158
|   DefectReleases <= 21.95122
|   |   LineDeletedIRLDE1 <= 17.518248: NO (222.67/35.0)
|   |   LineDeletedIRLDE1 > 17.518248
|   |   |   Releases <= 206: NO (34.33/10.0)
|   |   |   Releases > 206
|   |   |   |   RevisionAuthor <= 3.125: YES (7.0)
|   |   |   |   RevisionAuthor > 3.125: NO (3.0/1.0)
|   DefectReleases > 21.95122
|   |   DefectReleases <= 67.142857
|   |   |   AlterType = large
```

(a) No temporal features

```
ReportedIssues1Month <= 0: NO (3692.0/21.0)
ReportedIssues1Month > 0
|   Revision1Month <= 1: YES (105.0/2.0)
|   Revision1Month > 1
|   |   LineAddedIRLADD <= 3.636364: NO (16. 0/2.0)
|   |   LineAddedIRLADD > 3.636364
|   |   |   ReportedIssues <= 7: YES (15.0/1.0)
|   |   |   ReportedIssues > 7: NO (3.0)
```

(b) 1-month temporal features

```
DefectAppearance1Month <= 0
|   ReportedI5months <= 0: NO (3599.0/9.0)
|   ReportedI5months > 0
|   |   Revision2Months <= 4: NO (86.0/7.0)
|   |   Revision2Months > 4
|   |   |   LineAddedIRLADD <= 1.359223: NO (2.0)
|   |   |   LineAddedIRLADD > 1.359223: YES (5.0)
DefectAppearance1Month > 0
|   Revision1Month <= 1: YES (105.0/2.0)
|   Revision1Month > 1
```

(c) significant features

Figure 2: Top levels of induced defect location trees

Summarizing, we can say that the experiment for defect location prediction clearly shows that one *can, indeed, predict the location of bugs with a high accuracy.* We can also say that this accuracy bases (to a large extent) on temporal features. We hypothesize that one reason for the effectiveness of temporal values is that bugs usually survive more than one release. Other reasons might be the fact that complicated/complex or badly engineered classes are likely to exhibit bugs repeatedly unless they are re-engineered. Furthermore, we observe that the most important non-temporal features for prediction are the relation between line changes due to feature additions versus line changes due to bug fixing in the past – a type of feature not yet largely investigated in the literature, which clearly deserves more attention.

4.2 Predicting the Number of Bugs

The goal of the second group of experiments is to establish if our approach can amply predict the number of bugs that occur in any given file. This task is more difficult than the last, as it not only has to predict the existence of bugs (i.e., if $\#bugs > 0$) but the actual numbers of bugs. Since we believe that the task of predicting the number of bugs exhibits non-linear properties (a belief, for which we show evidence in section 4.3) we decided to use a non-linear regression approach. To preserve the comprehensibility of the model as well as the comparability of the model to the defect location prediction above we chose the Weka implementation of the M5 tree regression algorithm [14] called M5P. A regression tree model combines a decision tree an a linear regression by partitioning the feature space with a decision tree and then providing a linear regression equation for each of the tree's leafs. The model can, thus, predict a number by assigning any instance (i.e., entity to predict) to a leaf and then performing the associated regression to compute a number. This approach has the advantage that the regressions at the leafs do not have to be linearly connected – the tree provides the non-linear partition, the linear regressions predict the number.

The predictive power of temporal features. To test our proposition – that temporal features improve the prediction quality – we followed the same procedure as above: we learned the model with different base-sets of features either using no temporal data whatsoever (i.e., excluding features 8-19 of Table 2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months.

Table 5 summarizes the results for this comparison. Like Table 3 it Lists the name of the model, the features chosen by the feature selection algorithm and the root node of the regression tree. As performance measures it lists the Pearson correlation between the prediction and the actual data, the mean absolute error (MEA), and the root mean square error (RMSE). The results mostly mirror the ones form the location prediction experiments.

The models that can rely on the temporal features do so and even use it as the main feature for the decision tree. In contrast to the the location prediction, though, the root nodes of the tree do not have the most recent available number of reported issues or bugs (i.e., `ReportedI1Month`, `ReportedI2Month`, `ReportedI2Month`, `DefectAppearance1-Months`, or alternatively `DefectAppearance3Months`), but exclusively uses the number of available (i.e., `RevisionXMonth`, where `X` is the most recent available number for learning). While this is surprising at the surface further investigation clarifies the issue: when investigating the features chosen by the feature selection algorithm we can clearly see that the elements chosen as root nodes in the defect location prediction are used in the defect number prediction. In contrast to the defect location prediction they are not at the root of the partitioning decision tree but are mostly used in the regression function at the leafs. Consider, for example, the model induced for significant-features model as shown in Figure 3. At the top we can clearly see the decision tree that partitions the feature space using only some of the features. Below, the

Name of model	Features chosen	Root Node	Spearman's ρ	MAE	RMSE
no temporal features	3,5,7,20,21,22	`LineDeletedIRLDel`	0.863	0.0524	0.1898
1-months features	2,3,5,6,7,10,8,9,10,20,21,22	`Revision1Month`	0.941	0.0226	0.1272
2-months features	3,5,6,7,11,12,13,21,22	`Revision2Months`	0.950	0.0249	0.133
3-months features	2,5,7,14,15,16,21	`Revision3Months`	0.966	0.0241	0.1298
5-months features	2,3,5,17,18,19,22	`Revision5Months`	0.942	0.0326	0.1575
significant features	5,7,8,9,12,14,15,16,21,22	`Revision1Month`	0.963	0.0194	0.1119

Table 5: Results of different models for defect location prediction with M5P

figure shows the first of 8 linear regression models. this particular model is called if the rule at the root of the tree (`Revision1Month` $\leqslant 0.5$) is true. As the regression shows it uses the root node of the defect prediction decision tree `DefectAppearance1Month` with the second strongest weight in the regression.

```
Revision1Month <= 0.5 : LM1 (1348/0%)
Revision1Month >  0.5 :
|   LineAddedIRLADD <= 0.098 :
|   |   AgeMonths <= 33.667 :
|   |   |   Releases <= 65.5 : LM2 (343/0%)
|   |   |   Releases >  65.5 :
|   |   |   |   AgeMonths <= 15.95 : LM3 (112/87.619%)
|   |   |   |   AgeMonths >  15.95 : LM4 (266/26.955%)
|   |   AgeMonths >  33.667 : LM5 (975/0%)
|   LineAddedIRLADD >  0.098 :
|   |   Defectappearance3Months <= 0.5 : LM6 (619/42.644%)
|   |   Defectappearance3Months >  0.5 :
|   |   |   Revison3Months <= 1.5 : LM7 (81/171.567%)
|   |   |   Revison3Months >  1.5 : LM8 (87/210.532%)

LM num: 1
NumberofErroresLastMonth =
        0 * LineAddedIRLADD
        + 0 * AgeMonths
        + 0.0005 * Revison3Months
        - 0.0005 * Defectappearance3Months
        - 0.0013 * ReportedI3Months
        - 0 * Releases
        + 0 * RevisionAuthor
        - 0.0002 * Revision5Months
        + 0.0002 * DefectAppearance5Months
        - 0.0002 * Revision1Month
        + 0.0019 * DefectAppearance1Month
        + 0.0043 * ReportedI2Months
        - 0.0003
```

Figure 3: Excerpt of bug prediction model relying on significant features

Like in the bug prediction case Table 5 also clearly shows how the models with temporal features dominate the model without them. The difference in the Spearman's ρ (0.963 for temporal features vs. 0.863 without temporal features) is striking. The error rates MAE and RMSE mirror this behavior. Therefore, the results support our argument that the temporal data improve the accuracy of prediction model.

Exploring the prediction error. A closer look at the error rates in Table 5 also reveals that the RMSE is an order of magnitude larger than the MAE for all the models. This indicates that there are some large errors, which weigh in more heavily in the RMSE. Table 6 shows the histogram analysis of residual error of the significant-features model. As the table shows the bulk of the prediction has no (74.07%) or little (i.e., error $\leqslant 0.5$; in 98.69%). Nonetheless, a few predictions exhibit an error larger than 1. It is these predictions that mostly influence the error. When removing the file with a prediction error of 2.93, the MAE is lowered to 0.0194, but the RMSE is lowered to 0.0014, a full order of magnitude smaller. It is, hence, this one outlier that mostly contributes to the RSME. When, furthermore, removing all 5 files with an error larger than 1 we get a MAE of 0.0177 and a RMSE of 0.0095. We can, thus, conclude that the prediction error of our method is, in general, very small.

Error Interval	Frequency	Absolute	Cumulative
0	2838	74.08 %	74.08 %
$0 < e \leqslant 0.5$	943	24.61 %	98.69 %
$0.5 < e \leqslant 1$	45	1.17 %	99.87 %
$1 < e \leqslant 1.5$	4	0.10 %	99.97 %
$1.5 < e \leqslant 2$	0	0 %	99.97 %
$2 < e \leqslant 2.5$	0	0 %	99.97 %
$2.5 < e \leqslant 3$	1	0.03 %	100.00 %

Table 6: Residual error histogram for significant-feature model

Comparison with other defect predictions using the same data set The MSR Mining Challenge 2007,[4] which provided the data we used for our study, had a similar task as its Challenge #2. The main difference between our approach and the challenge task is that we chose to make our predictions on the file level and the Challenge task required participants to predict the number of bugs for 32 plug-ins[5] (i.e., summarizing the bugs for all their classes). Two methods, one in two versions were submitted to the mining challenge. `C-ESSEN` by Adrian Schröter [15] predicted the bugs based on the import statements used in the files. This is a measure we did not use at all. ULAR by Joshi et. al [4] uses features computed in the last month to make a prediction for next month. A second version of ULAR extends those predictions with a trend analysis. Last, an ad-hoc method used as an comparison by Thomas Zimmermann (called 1 Year ago) simply takes the measures from 2006 to make the prediction for 2007. Table 7 shows the Spearman's rank correlation (ρ) for all the methods as well as our significant-feature model. The results clearly show that our approach is better at ranking the files according to their expected bugs. The ranking, rather than the precise prediction of the number of bugs, is actually an important task when one tries to make an optimal assignment of resources (i.e., programmers) to tasks (i.e., the fixing of bugs) [12]. Note, however, that the other models are making their prediction on 32 modules whereas we limit ourselves to only 6, which is a much simpler task.

[4] http://msr.uwaterloo.ca/msr2007/challenge/

[5] http://msr.uwaterloo.ca/msr2007/challenge/plugins.txt

Model	n	ρ
C-ESSEN (imports) [15]	32	0.67
ULAR (Last month + trends) [4]	32	0.81
ULAR (Last months) [4]	32	0.84
1 Year Ago	32	0.91
significant-features model	6	1.00

Table 7: Spearman's ρ for MSR Mining Challenge 2007 results, where n is the number of components

Summarizing, we can say that our non-linear bug prediction model supports our proposition that temporal features are imperative for an accurate prediction – without them the Spearman's rank correlation ρ between the predictions and the actual error numbers is lowered from 0.963 to 0.863. Second, we can clearly see how our model is highly accurate for most predictions and that most of the residual error is introduced by 5 predictions of 3831. Third, we tried to compare the performance of our approach to similar tasks (as we did not find any work on the same task): We find that our approach exhibits a superior performance compared to others with respect to the Spearman's rank correlation.

4.3 The Predictive power of linear and non-linear prediction methods

The second of our guiding propositions is that non-linear models should provide a superior prediction that the usually used linear ones. Specifically, we stated that the non-linear models are able to exploit the non-linear relationships between the features to make more accurate bug number predictions. By non-linear we mean here a relationship that cannot be captured by a weighted sum of simple, continuous functions of the single features (as done by a linear regression), but may require functions of two or more features. To explore this hypothesis we re-ran experiments outlined in sub-section 4.2 with a standard linear regression algorithm.

Table 8 shows the results of this analysis comparing the Spearman's rank correlation (ρ), the mean absolute error (MAE), and the root mean squared error (RSME) for the linear model (LM) – a standard linear regression – and the non-linear model in the form of the M5P algorithm. The results show that the non-linear significantly outperforms the linear model for all performance measures (results for pairwise t-test significant at: $p = 1.09\%$ for ρ, $p = 0.29\%$ for MEA, and $p = 2.42\%$ for RMSE). The dominance is, however, not constant. For the data sets without temporal features the LM and M5P have a very similar performance. The more recent temporal features the more pronounced is the dominance of the non-linear model. This would lead us to hypothesize that the non-temporal features exhibit a non-linear relationship to the number of bugs. If we explore the actual model this hypothesis is confirmed. Consider again Figure 3, which shows the bug prediction model for significant-features. As the model clearly shows the temporal features are heavily used within the non-linear element of the model: the decision tree that partitions the feature space. Nonetheless, the temporal features are also reused in the linear part of the model: the leaf-based regressions. We can, thus, conclude that (1) the temporal features have both linear and non-linear elements with respect to the number of bugs and (2) the M5P's capability to exploit both linear and non-linear elements clearly results in more accurate results.

4.4 Identifying the critical Eclipse plugins

We applied the best performing prediction model to identify the most critical Eclipse plugins (out of the six). These plugins need be considered first when planning refactoring and testing efforts. With critical we mean plugins for which our model predicts the highest number of bugs for January 2007. Table 9 lists the results with the actual number of bugs, the predicted number of bugs, and the accuracy of the prediction model.

Pugin	#Actual	#Predicted	Accuracy
pdeui	83	68.8999	83.0119%
compare	36	29.5561	82.1002%
pdebuild	20	16.7421	83.7106%
updateui	10	8.6371	86.3718%
updatecore	8	7.1928	89.9104%
search	1	1.0663	93.7836%

Table 9: Predicted and actual number of bugs for the six Eclipse plugins in January 2007.

From a managers point of view the number of predicted bugs clearly indicates that refactoring as well as testing effort needs to be dedicated to the two plugins `pdeui` and `compare`. In particular, `pdeui` is indicated as a critical plugin that according to our model will be affected by around 69 bugs in January 2007. This mirrors the actual number of bugs, which was 83. From that we conclude that such predictions provide a valuable input for software project managers to plan refactorings and tests.

5. LIMITATIONS AND CONCLUSIONS

Our findings for the Eclipse evolution data are very promising. The use of a non-linear model basing on temporal features selected by an automated feature selection algorithm could predict defect location and numbers with a very high accuracy. The findings are, however, hampered with a number of limitations.

First and foremost, the chosen *Eclipse data set represents only one project-family.* While we followed good data mining practice to ensure the generalizability of the our findings the data might behave Eclipse idiosyncratic such as a common culture of bug-reporting or code documentation/fixing, programming language dependencies (Eclipse only uses Java), etc. Furthermore, we only looked at the predictions for the last month (January 2007) of the data set. We intend ascertain the generalizability of our findings by (1) exploring the quality of the prediction for other months within the Eclipse data set and for other projects altogether.

Second, we *"only" used one off-the-shelf feature selection and non-linear induction algorithm.* It might, therefore, be that the resulting feature set and model are suboptimal. Following good data analysis practice we should try a whole set of algorithms to determine the most predictive model – a task that we will undertake in the near future. Nonetheless, we are confident that the use of other algorithms will not substantially change our findings. Much more we expect them to potentially make them even more pronounced than currently.

Third, our candidate features were chosen by our study of

Name of model	LM ρ	LM MAE	LM RMSE	M5P ρ	M5P MAE	M5P RMSE
without temporal features	0.844	0.0569	0.1902	0.863	0.0524	0.1898
1-Month	0.935	0.0306	0.1311	0.941	0.0226	0.1272
2-Months	0.919	0.039	0.1421	0.950	0.0249	0.133
3-Months	0.891	0.0471	0.1523	0.966	0.0241	0.1298
5-Months	0.918	0.0423	0.1611	0.942	0.0326	0.1575
Significant Features	0.929	0.0319	0.1227	0.963	0.0194	0.1119

Table 8: Comparison of linear model (LM) and Non-linear model (M5P), ρ is the Spearman's rank corr.

the literature and some of our own thoughts regarding temporal features. In order to ensure an optimal performance of the resulting models we need to *explore the full space of possibly applicable measures* (or features) reported in the literature. We hope to investigate the full feature space in the future. Like with the feature selection, however, we think that such an exploration would make our finding more pronounced but not change the inferred conclusions.

Last and most importantly, our attempt could be seen as a post-prediction rather than a pure prediction. After all, we could employ some "current" information in building our models. We intend to address this problem in the future by completely temporally disentangling training from test set. In the future we intend to *embed this approach into a tool, which seamlessly integrates into an IDE* and highlights files that have a high probability of defects or a large number of bugs. Such an integration would simplify the use of the algorithm by software managers and developers, which would allow to investigate their use in practice.

We also intend to pursue the avenue of temporal dependencies/relationship between code/bug-measures and future performance. To that end we also intend to explore the use of temporal data mining techniques such a Markov models. In closing we should highlight that *our approach - employing temporal features and non-linear models for defect prediction shows a clear advantage over others*. We hope that this method will help to contribute to improved bug number predictions and, therefore, help to ensure the development of software with fewer bugs.

6. ACKNOWLEDGMENTS

This work was partially supported by a grant of the Sri Lankan government. We would like to thank Thomas Zimmermann for making available the data for the 2007 MSR mining challenge as well as the anonymous reviewers whose comments helped to improve this paper.

7. REFERENCES

[1] M. Askari and R. Holt. Information theoretic evaluation of change prediction models for large-scale software. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 126–132, New York, NY, USA, 2006. ACM Press.

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

[3] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[4] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak. Local and global recency weighting approach to bug prediction. In *MSR 2007: International Workshop on Mining Software Repositories*, 2007.

[5] T. M. Khoshgoftaar, E. B. 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 Seventh International Symposium on Software Reliability Engineering*, pages 364–371, White Plains, NY, 1996. IEEECS.

[6] C. Kiefer, A. Bernstein, and J. Tappolet. Analyzing software with isparql. In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007)*. Springer, June 2007. to appear.

[7] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, New York, NY, USA, 2006. ACM Press.

[8] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

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

[10] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering, p580–586*, 2005.

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

[12] F. J. Provost and T. Fawcett. Robust classification for imprecise environments. volume 42, pages 203–231, 2001.

[13] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[14] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992.

[15] A. Schröter. Predicting defects and changes with import relations. In *Proceedings of MSR 2007: International Workshop on Mining Software Repositories*, 2007.

[16] J. Sliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness (research demonstration). In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107–110. ACM, September 2005.

[17] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, second edition, 2005.

[18] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse, May 2007.