Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction

Emanuel Giger, Martin Pinzger, and Harald C. Gall

Report TUD-SERG-2011-007
Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction

Emanuel Giger
Department of Informatics
University of Zurich
giger@ifi.uzh.ch

Martin Pinzger
Department of Software Engineering
Delft University of Technology
m.pinzger@tudelft.nl

Harald C. Gall
Department of Informatics
University of Zurich
gall@ifi.uzh.ch

ABSTRACT
A significant amount of research effort has been dedicated to learning prediction models that allow project managers to efficiently allocate resources to those parts of a software system that most likely are bug-prone and therefore critical. Prominent measures for building bug prediction models are product measures, e.g., complexity or process measures, such as code churn. Code churn in terms of lines modified (LM) and past changes turned out to be significant indicators of bugs. However, these measures are rather imprecise and do not reflect all the detailed changes of particular source code entities during maintenance activities. In this paper, we explore the advantage of using fine-grained source code changes (SCC) for bug prediction. SCC captures the exact code changes and their semantics down to statement level. We present a series of experiments using different machine learning algorithms with a dataset from the Eclipse platform to empirically evaluate the performance of SCC and LM. The results show that SCC outperforms LM for learning bug prediction models.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—performance measures, process measures, software science

General Terms
Management, Measurement, Reliability, Experimentation

Keywords
Software bugs, code churn, source code changes, prediction models, nonlinear regression

1. INTRODUCTION
Bugs in software systems are a key risk and major cost driver for both, companies that develop software and companies that consume software systems in their daily business. Development teams are typically exposed to time pressure and costs. Often, Quality Assurance (QA) suffers from these constraints, and project managers are forced to allocate their limited resources with maximum efficiency. Research has developed bug prediction models that help managers in a structured manner to allocate QA resources to those parts of a system that likely contain most of the bugs rather than relying solely on their experience.

Prominent measures for building bug prediction models are product measures, e.g., complexity [26], or process measures, such as code churn [15]. Prior work found out that process measures perform explicitly well [24]. However, existing measures such as code churn based on lines modified (LM) suffer from the fact that they do not capture the semantics of code changes. For example, the source file BinaryCompareViewerCreator.java of the Eclipse plugin Compare had 8 revisions and in total 81 lines were changed. None of these changes affected any source code entity since only license header or indentation updates have been performed. Fine-grained source code changes (SCC) as introduced by Fluri et al. [13] on the other hand capture the semantics of changes. For example, between revision 1.1 and 1.2 of the source file CompareEditorSelectionProvider.java of the same plugin a single if-statement was removed. Between revisions 1.2 and 1.3 a nested if-statement was added, import statements were updated, and two methods were added.

In our previous work, we pointed out this discrepancy between changes based on a text-line level and fine-grained source code changes, and showed that fine-grained source code changes can be used to qualify change couplings between source files [11].

In this paper, we explore with a series of prediction experiments using data from the Eclipse platform how SCC relates to bugs and to what extent bug prediction models benefit from having more detailed information about source code changes. In particular, we investigate the following three research hypotheses:

\[ H_1: \text{SCC does have a stronger correlation with the number of bugs than LM.} \]

\[ H_2: \text{SCC achieves better performance to classify source files into bug- and not bug-prone files than LM.} \]

\[ H_3: \text{SCC achieves better performance when predicting the number of bugs in source files than LM.} \]

The results of our study with Eclipse projects show that SCC significantly outperforms LM for learning bug prediction models to classify source files into bug- and not bug-prone, as well as to predict the number of bugs in source files.

The remainder of this paper is organized as follows: In Section 2, we give an overview of our approach and outline the steps to prepare the data. Section 3 presents the empirical
study with the Eclipse projects. We discuss our findings in Section 4 and threats to validity in Section 5. In Section 6, we present related work and then draw our conclusions in Section 7.

2. APPROACH

In this section, we describe the methods and tools we used to extract and preprocess the data (see Figure 1). Basically, we take into account three main pieces of information about the history of a software system to assemble the dataset for our experiments: (1) versioning data including lines modified (LM), (2) bug data, i.e., which files contained bugs and how many of them (Bugs), and (3) fine-grained source code changes (SCC).

1. Versioning Data. We use EVOLIZER [14] to access the versioning repositories, e.g., CVS, SVN, or GIT. They provide log entries that contain information about revisions of files that belong to a system. From the log entries we extract the revision number (to identify the revisions of a file in correct temporal order), the revision timestamp, the name of the developer who checked-in the new revision, and the commit message. We then compute LM for a source file as the sum of lines added, lines deleted, and lines changed per file revision.

2. Bug Data. Bug reports are stored in bug repositories such as Bugzilla. Traditional bug tracking and versioning repositories are not directly linked. We first establish these links by searching references to reports within commit messages, e.g., "fix for 12023" or "bug#23467". Prior work used this method and developed advanced matching patterns to catch those references [10, 33, 39]. Again, we use EVOLIZER to automate this process. We take into account all references to bug reports. Based on the links we then count the number of bugs (Bugs) per file revision.

3. Fine-Grained Source Code Changes (SCC): Current versioning systems record changes solely on file level and textual basis, i.e., source files are treated as pure text files. In [11], Fluri et al. showed that LM recorded by versioning systems might not accurately reflect changes in the source code. For instance, source formatting or license header updates generate additional LM although no source code entities were changed; changing the name of a local variable and a method likely result both in "1 line changed" but are different modifications. Fluri et al. developed a tree differencing algorithm for fine-grained source code change extraction [13]. It allows to track fine-grained source changes down to the level of single source code statements, e.g., method invocation statements, between two versions of a program by comparing their respective abstract syntax trees (AST). Each change then represents a tree edit operation that is required to transform one version of the AST into the other. The algorithm is implemented in CHANGE DISTILLER [14] that pairwise compares the ASTs between all direct subsequent revisions of each file. Based on this information, we then count the number of different source code changes (SCC) per file revision.

The preprocessed data from step 1-3 is stored into the Release History Database (RHDB) [10]. From that data, we then compute LM, SCC, and Bugs for each source file by aggregating the values over the given observation period.

3. EMPIRICAL STUDY

In this section, we present the empirical study that we performed to investigate the hypotheses stated in Section 1. We discuss the dataset, the statistical methods and machine learning algorithms we used, and report on the results and findings of the experiments.

3.1 Dataset and Data Preparation

We performed our experiments on 15 plugins of the Eclipse platform. Eclipse is a popular open source system that has been studied extensively before [4, 27, 38, 39].

Table 1 gives an overview of the Eclipse dataset used in this study with the number of unique *.java files (Files), the total number of java file revisions (Rev.), the total number of lines added, deleted, and changed (LM), the total number of fine-grained source code changes (SCC), and the total number of bugs (Bugs) within the given time period (Time). Only source code files, i.e., *.java, are considered.

After the data preparation step, we performed an initial analysis of the extracted SCC. This analysis showed that there are large differences of change type frequencies, which might influence the results of our empirical study. For instance, the change types Parent Class Delete, i.e., removing a super class from a class declaration, or Removing Method Overridability, i.e., adding the java keyword final to a method declaration, are relatively rare change types. They constitute less than one thousandth of all SCC in the entire study corpus. Whereas one fourth of all SCC are Statement Insert changes, e.g., the insertion of a new local variable declaration. We therefore aggregate SCC according to their change type semantics into 7 categories of SCC for our further analysis. Table 2 shows the resulting aggregated categories and their respective meanings.
Table 2: Categories of fine-grained source code changes

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cDecl</td>
<td>Aggregates all changes that alter the declaration of a class: Modifier changes, class renaming, class API changes, parent class changes, and changes in the &quot;implements list&quot;.</td>
</tr>
<tr>
<td>oState</td>
<td>Aggregates the insertion and deletion of object states of a class, i.e., adding and removing fields.</td>
</tr>
<tr>
<td>func</td>
<td>Aggregates the insertion and deletion of functionality of a class, i.e., adding and removing methods.</td>
</tr>
<tr>
<td>mDecl</td>
<td>Aggregates all changes that alter the declaration of a method: Modifier changes, method renaming, method API changes, return type changes, and changes of the parameter list.</td>
</tr>
<tr>
<td>stmt</td>
<td>Aggregates all changes that modify executable statements, e.g., insertion or deletion of statements.</td>
</tr>
<tr>
<td>cond</td>
<td>Aggregates all changes that alter condition expressions in control structures.</td>
</tr>
<tr>
<td>else</td>
<td>Aggregates the insertion and deletion of else-parts.</td>
</tr>
</tbody>
</table>

Table 3: Relative frequencies of SCC categories per Eclipse project, plus their mean and variance over all selected projects.

<table>
<thead>
<tr>
<th>Eclipse Project</th>
<th>cDecl</th>
<th>oState</th>
<th>func</th>
<th>mDecl</th>
<th>stmt</th>
<th>cond</th>
<th>else</th>
<th>Mean</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>0.01</td>
<td>0.06</td>
<td>0.28</td>
<td>0.05</td>
<td>0.24</td>
<td>0.03</td>
<td>0.03</td>
<td>0.01</td>
<td>0.0000</td>
</tr>
<tr>
<td>jFace</td>
<td>0.02</td>
<td>0.04</td>
<td>0.08</td>
<td>0.11</td>
<td>0.70</td>
<td>0.02</td>
<td>0.03</td>
<td>0.01</td>
<td>0.0000</td>
</tr>
<tr>
<td>JDT Debug</td>
<td>0.02</td>
<td>0.06</td>
<td>0.08</td>
<td>0.10</td>
<td>0.70</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0000</td>
</tr>
<tr>
<td>Resource</td>
<td>0.01</td>
<td>0.04</td>
<td>0.02</td>
<td>0.11</td>
<td>0.77</td>
<td>0.03</td>
<td>0.02</td>
<td>0.01</td>
<td>0.0000</td>
</tr>
<tr>
<td>Runtime</td>
<td>0.01</td>
<td>0.05</td>
<td>0.07</td>
<td>0.10</td>
<td>0.73</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
<td>0.0000</td>
</tr>
<tr>
<td>Team Core</td>
<td>0.05</td>
<td>0.04</td>
<td>0.13</td>
<td>0.17</td>
<td>0.57</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0000</td>
</tr>
<tr>
<td>CVS Core</td>
<td>0.01</td>
<td>0.04</td>
<td>0.10</td>
<td>0.07</td>
<td>0.73</td>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>Debug Core</td>
<td>0.04</td>
<td>0.07</td>
<td>0.02</td>
<td>0.13</td>
<td>0.69</td>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>JFace Text</td>
<td>0.04</td>
<td>0.03</td>
<td>0.06</td>
<td>0.11</td>
<td>0.70</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>Update Core</td>
<td>0.02</td>
<td>0.04</td>
<td>0.07</td>
<td>0.09</td>
<td>0.74</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0000</td>
</tr>
<tr>
<td>Debug UI</td>
<td>0.02</td>
<td>0.06</td>
<td>0.09</td>
<td>0.07</td>
<td>0.70</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>JDT Debug UI</td>
<td>0.01</td>
<td>0.07</td>
<td>0.07</td>
<td>0.05</td>
<td>0.75</td>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>Help</td>
<td>0.02</td>
<td>0.05</td>
<td>0.08</td>
<td>0.07</td>
<td>0.73</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.0000</td>
</tr>
<tr>
<td>JDT Core</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>0.05</td>
<td>0.80</td>
<td>0.05</td>
<td>0.04</td>
<td>0.04</td>
<td>0.0000</td>
</tr>
<tr>
<td>OSGI</td>
<td>0.03</td>
<td>0.04</td>
<td>0.06</td>
<td>0.11</td>
<td>0.71</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0000</td>
</tr>
<tr>
<td>Mean</td>
<td>0.02</td>
<td>0.05</td>
<td>0.07</td>
<td>0.09</td>
<td>0.72</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.0000</td>
</tr>
<tr>
<td>Variance</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0003</td>
<td>0.0005</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

Some change types defined in [11] such as the ones that change the declaration of an attribute are left out in our analysis as their total frequency is below 0.8%. The complete list of all change types, their meanings and their contexts can be found in [11].

Table 3 shows the relative frequencies of each category of SCC per Eclipse project, plus their mean and variance over all selected projects. Looking at the mean values listed in the second last row of the table, we can see that 70% of all changes are stmt changes. These are relatively small changes and affect only single statements. Changes that affect the existing control flow structures, i.e., cond and else, constitute only about 6% on average. While these changes might affect the behavior of the code, their impact is locally limited to their proximate context and blocks. They ideally do not induce changes at other locations in the source code. cDecl, oState, func, and mDecl represent about one fourth of all changes in total. They change the interface of a class or a method and do—except when adding a field or a method—require a change in the dependent classes and methods. The impact of these changes is according to the given access modifiers; within the same class or package (private or default) or external code (protected or public).

The values in Table 3 show small variances and relatively narrow confidence intervals among the categories across all projects. This is an interesting observation as these Eclipse projects do vary in terms of file size and changes (see Table 1).

3.2 Correlation of SCC Categories

We first performed a correlation analysis between the different SCC categories of all source files of the selected projects. We use the Spearman rank correlation because it makes no assumptions about the distributions, variances and the type of relationship. It compares the ordered ranks of the variables to measure a monotonic relationship. This makes Spearman more robust than Pearson correlation, which is restricted to measure the strength of a linear association between two normal distributed variables [8]. Spearman values of +1 and -1 indicate a high positive or negative correlation, whereas 0 tells that the variables do not correlate at all. Values greater than +0.5 and lower than -0.5 are considered to be substantial; values greater than +0.7 and lower than -0.7 are considered to be strong correlations [31].

Table 4 lists the results. Some facts can be read from the values: cDecl does neither have substantial nor strong correlation with any of the other change types. oState has its highest correlation with func. func has approximately equal high correlations with oState, mDecl, and stmt. The strongest correlations are between stmt, cond, and else with 0.71, 0.7, and 0.67.

While this correlation analysis helps to gain knowledge about the nature and relation of change type categories it mainly reveals multicollinearity between those categories that we have to address when building regression models. A causal interpretation of the correlation values is tedious and must be dealt with caution. Some correlations make sense and could be explained using common knowledge about programming. For instance, the strong correlations between stmt, cond, and else can be explained that often local variables are affected when existing control structures are changed. This is because they might be moved into a new else-part or because a new local variable is needed to handle the different conditions. In [12], Fluri et al. attempt to find an explanation why certain change types occur more frequently together than others, i.e., why they correlate.

3.3 Correlation of Bugs, LM, and SCC

H1 formulated in Section 1 aims at analyzing the correlation between Bugs, LM, and SCC (on the level of source files). It serves two purposes: (1) We analyze whether there is a significant correlation between SCC and Bugs. A significant correlation is a precondition for any further analysis and prediction model. (2) Prior work reported on the positive relation between Bugs and LM. We explore the extent to which SCC has a stronger correlation with Bugs than LM. We apply the Spearman rank correlation to each selected Eclipse project to investigate H1.
Table 5: Spearman rank correlation between Bugs and LM, SCC, and SCC categories (\*marks significant correlations at \( \alpha = 0.01 \)).

<table>
<thead>
<tr>
<th>Eclipse Project</th>
<th>LM</th>
<th>SCC</th>
<th>cDecl</th>
<th>oState</th>
<th>func</th>
<th>mDecl</th>
<th>stmt</th>
<th>cond</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>0.88</td>
<td>0.76</td>
<td>0.74</td>
<td>0.61</td>
<td>0.67</td>
<td>0.61</td>
<td>0.66</td>
<td>0.55</td>
<td>0.52</td>
</tr>
<tr>
<td>jFace</td>
<td>0.74</td>
<td>0.71</td>
<td>0.41</td>
<td>0.47</td>
<td>0.57</td>
<td>0.63</td>
<td>0.66</td>
<td>0.51</td>
<td>0.48</td>
</tr>
<tr>
<td>Resource</td>
<td>0.75</td>
<td>0.86</td>
<td>0.49</td>
<td>0.62</td>
<td>0.73</td>
<td>0.73</td>
<td>0.67</td>
<td>0.49</td>
<td>0.46</td>
</tr>
<tr>
<td>Team Core</td>
<td>0.15</td>
<td>0.66</td>
<td>0.44</td>
<td>0.43</td>
<td>0.56</td>
<td>0.52</td>
<td>0.53</td>
<td>0.36</td>
<td>0.35</td>
</tr>
<tr>
<td>CVS Core</td>
<td>0.60</td>
<td>0.79</td>
<td>0.39</td>
<td>0.62</td>
<td>0.66</td>
<td>0.57</td>
<td>0.72</td>
<td>0.58</td>
<td>0.54</td>
</tr>
<tr>
<td>Debug Core</td>
<td>0.63</td>
<td>0.78</td>
<td>0.45</td>
<td>0.55</td>
<td>0.61</td>
<td>0.51</td>
<td>0.59</td>
<td>0.45</td>
<td>0.46</td>
</tr>
<tr>
<td>Runtime</td>
<td>0.66</td>
<td>0.79</td>
<td>0.47</td>
<td>0.58</td>
<td>0.66</td>
<td>0.61</td>
<td>0.66</td>
<td>0.55</td>
<td>0.54</td>
</tr>
<tr>
<td>JDT Debug</td>
<td>0.62</td>
<td>0.80</td>
<td>0.42</td>
<td>0.45</td>
<td>0.56</td>
<td>0.55</td>
<td>0.64</td>
<td>0.46</td>
<td>0.44</td>
</tr>
<tr>
<td>jFace Test</td>
<td>0.75</td>
<td>0.74</td>
<td>0.50</td>
<td>0.55</td>
<td>0.54</td>
<td>0.64</td>
<td>0.62</td>
<td>0.59</td>
<td>0.55</td>
</tr>
<tr>
<td>JDT Debug UI</td>
<td>0.80</td>
<td>0.81</td>
<td>0.46</td>
<td>0.57</td>
<td>0.62</td>
<td>0.53</td>
<td>0.74</td>
<td>0.57</td>
<td>0.54</td>
</tr>
<tr>
<td>Update Core</td>
<td>0.43</td>
<td>0.62</td>
<td>0.63</td>
<td>0.4</td>
<td>0.43</td>
<td>0.51</td>
<td>0.45</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td>Debug UI</td>
<td>0.56</td>
<td>0.61</td>
<td>0.44</td>
<td>0.50</td>
<td>0.63</td>
<td>0.60</td>
<td>0.72</td>
<td>0.54</td>
<td>0.52</td>
</tr>
<tr>
<td>Help</td>
<td>0.54</td>
<td>0.48</td>
<td>0.37</td>
<td>0.43</td>
<td>0.42</td>
<td>0.43</td>
<td>0.44</td>
<td>0.36</td>
<td>0.41</td>
</tr>
<tr>
<td>JDT Core</td>
<td>0.70</td>
<td>0.74</td>
<td>0.39</td>
<td>0.6</td>
<td>0.69</td>
<td>0.70</td>
<td>0.67</td>
<td>0.62</td>
<td>0.6</td>
</tr>
<tr>
<td>OSGI</td>
<td>0.70</td>
<td>0.77</td>
<td>0.47</td>
<td>0.6</td>
<td>0.66</td>
<td>0.65</td>
<td>0.63</td>
<td>0.57</td>
<td>0.48</td>
</tr>
<tr>
<td>Mean</td>
<td>0.62</td>
<td>0.74</td>
<td>0.46</td>
<td>0.53</td>
<td>0.6</td>
<td>0.59</td>
<td>0.63</td>
<td>0.51</td>
<td>0.48</td>
</tr>
<tr>
<td>Median</td>
<td>0.66</td>
<td>0.70</td>
<td>0.45</td>
<td>0.55</td>
<td>0.62</td>
<td>0.60</td>
<td>0.66</td>
<td>0.54</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table 5 lists the results of the correlation analysis per project. The second and third columns on the left hand side show the correlation values between Bugs and LM, and total SCC. The values for LM show that except for two projects all correlations are at least substantial, some are even strong. The mean of the correlation is 0.62 and the median is 0.66. This indicates that there is a substantial, observable positive correlation between LM and bugs meaning that an increase in LM leads to an increase in bugs in a source file. This result confirms previous research presented in [15,25,27].

The values in the third column show that all correlations for SCC are positive and most of them are strong. The mean of the correlation is 0.74 and the median is 0.77. Some Eclipse projects show correlation values of 0.8 and higher. Two values are below 0.7 and only one is slightly lower than 0.5. All values are statistically significant. This denotes an overall strong correlation between Bugs and SCC that is even stronger than between Bugs and LM. We applied a One Sample Wilcoxon Signed-Ranks Test on the SCC correlation values against the hypothesized limits of 0.5c (substantial) and 0.7c (strong). They were significant at \( \alpha = 0.05 \). Therefore we conclude that there is a significant strong correlation between Bugs and SCC.

We further compared the correlation values of LM and SCC in Table 5 to test whether the observed difference is significant. On average, the correlation between Bugs and SCC is 0.12 stronger than the correlation between Bugs and LM. In particular, 12 out of 15 cases show a stronger correlation towards SCC with an average difference of 0.16. In some cases the differences are even more pronounced, e.g., 0.51 for Team Core or 0.25 for Debug Core. Other projects experience smaller differences such as 0.01 for JDT Debug UI and jFace, and 0.04 for JDT Core. Only in three cases the correlation of LM is stronger. The largest difference is 0.06 for Eclipse Help.

We applied a Related Samples Wilcoxon Signed-Ranks Test to test the significance of the correlation differences between LM and SCC. The rationale for such a test is that (1) we calculated both correlations for each project resulting in a matched correlation pair per project and (2) we can relax any assumption about the distribution of the values. The test was significant at \( \alpha = 0.05 \) rejecting the null hypothesis that the two medians are the same. Based on this result we can accept H1—SCC does have a stronger correlation with Bugs than LM.

As part of investigating H1, we also analyzed the correlation between bugs and the SCC categories we have defined in Table 2 to answer the question whether there are differences in how change types correlate with bugs.

The columns 4-10 on the right hand side of Table 5 show the correlations between the different categories and bugs for each Eclipse project. Regarding their mean, the categories stmt, func, and mDecl show the strongest correlation with Bugs. For some projects their correlation values are close or above 0.7, e.g., func for Resource or JDT Core; mDecl for Resource and JDT Core; stmt for JDT Debug UI and Debug UI. oState and cond still have a substantial correlation with the number of bugs indicated by an average correlation value of 0.53 and 0.51. cDecl and else have means below 0.5. This indicates that SCC categories do correlate differently with the number of bugs in our dataset.

To test whether this assumption holds, we first performed a Related Samples Friedman Test. The result was significant at \( \alpha = 0.05 \), so we can reject the null hypothesis that the distribution of the correlation values of SCC categories, i.e., the rows on the right hand side in Table 5 are the same. The Friedman Test operates on the mean ranks of related groups. We used this test because we repeatedly measured the correlations of the different categories on the same dataset, i.e., our related groups, and because it does not make any assumption about the distribution of the data and the sample size.

A Related Samples Friedman Test is a global test that only tests whether all of the groups differ. It does not tell anything between which groups the difference occurs. To test whether some pairwise groups differ stronger than others or do not differ at all post-hoc tests are required. We performed a Wilcoxon Test and Friedman Test on each pair including \( \alpha \)-adjustment.

The results showed two groups of SCC categories whose correlation values are not significantly different among each other: (1) else, cond, oState, and cDecl, and (2) stmt, func, and mDecl. The difference of correlation values between these groups is significant.

In summary, we found strong positive correlation between SCC and Bugs that is significantly stronger than the correlation between LM and Bugs. This indicates that SCC exhibits good predictive power, therefore we accepted H1. Furthermore, we observed a difference in the correlation values between several SCC categories and Bugs.

### 3.4 Predicting Bug- & Not Bug-Prone Files

The goal of H2 is to analyze how SCC performs compared to LM when discriminating between bug-prone and not bug-prone files in our dataset. We built models based on different machine learning techniques (in the following also called classifiers) and evaluated them with our Eclipse dataset.

Prior work states that some machine learning techniques perform better than others. For instance, Lessman et al. found out with an extended set of various classifiers that Random Forest performs the best on a subset of the NASA Metrics dataset [20]. But in return they state as well that performance differences between classifiers are marginal and not necessarily significant.

For that reason we used the following classifiers: Logistic Regression (LR), J48 (C 4.5 Decision Tree), RandomForest (RF), Bayesian Network (BNet) implemented by the WEKA toolkit [35], Exhaustive CHAID, a Decision Tree based on chi squared criterion by SPSS 18.0, Support Vector Machine (Lib-
SVM) [7], Naive Bayes Network (NBayes) and Neural Nets (NN) both provided by the Rapid Miner toolkit [23]. The classifiers calculate and assign a probability to each source file to be classified either into bug-prone or not-bug-prone.

For each Eclipse project, we binned files into bug-prone and not-bug-prone using the median of the number of bugs per file:

$$\text{bugClass} = \begin{cases} 
\text{not bug – prone} & : \text{Bugs} \leq \text{median} \\
\text{bug – prone} & : \text{Bugs} > \text{median}
\end{cases}$$

When using the median as cut point the labeling of a file is relative to how much bugs other files have in a project. This resulted in an average 57:43 prior probability towards not-bug-prone file in our dataset. There exist several ways of binning files afore. They mainly vary in that they result in different prior probabilities. For instance Zimmerman et al. [39] and Bernstein et al. [4] labeled files as bug-prone if they had at least one bug. When having heavily skewed distributions this approach may lead to a high prior probability towards one class. Nagappan et al. [27] used a statistically lower confidence bound. The different prior probabilities make the use of accuracy as a performance measure for classification difficult. As proposed in [20, 22], we therefore use the area under the receiver operating characteristic curve (AUC) as performance measure. AUC is independent of prior probabilities and therefore a robust measure to assess and compare the performance of predictor models [4]. AUC can be seen as the probability that a trained model assigns a higher score to the bug-prone file when choosing randomly a bug-prone and a not-bug-prone file [16]. We mainly use AUC for discussing and comparing the performance of prediction models. In addition, we also report on precision (P) and recall (R) to facilitate the comparison with existing work.

We performed two experiments to investigate H2: In Experiment 1 (E1), we used logistic regression once with total number of LM and once with number of SCC per file as predictors. In Experiment 2 (E2), we used the above mentioned classifiers and SCC categories as predictors to investigate whether the additional information about the change type category can improve the performance of classification models. In the following we discuss the results of both experiments by means of the AUC measure.

**Experiment 1:** Table 6 lists the AUC values of E1 for each project in our dataset. The models were validated using 10 fold cross validation, and the performance measures were computed when reapplying the prediction model to the dataset it was obtained from. Overall denotes the AUC of the model that was learned when merging all files of the projects into one larger dataset. SCC shows a lower performance than SCC (see column AUC_{LM}). Help is the only project where LM is a better predictor than SCC. This is not surprising as it is the project that yields the largest difference in correlation in favor of LM, see Table 5. In general, the correlation values in Table 5 reflect the picture given by the AUC values. For instance, jFace, jFace Text, and JDT Debug UI that exhibit similar correlations performed nearly equal. A Related Samples Wilcoxon Signed-Ranks Test on the AUC values of LM and SCC was significant at $\alpha = 0.05$: Logistic regression based on SCC is not only a good predictor but is a significant better predictor than LM to classify source files of Eclipse projects into bug-prone or not-bug-prone. Therefore, we can accept H2—SCC achieves better performance when discriminating between bug- and not-bug-prone files than LM.

**Experiment 2:** Table 7 lists the AUC values of each classifier for each project in our dataset. Analogously to E1, the values for AUC, precision, and recall were computed when reapplying the prediction model to the dataset it was obtained from (we skip the values for precision and recall for readability and space reasons). As before, the models were validated using 10 fold cross validation. Overall denotes the AUC of the model that was learned when merging all files of the projects into one larger dataset. When using logistic regression, multicollinearity between multiple predictors

<table>
<thead>
<tr>
<th>Eclipse Project</th>
<th>AUC_{LM}</th>
<th>AUC_{SCC}</th>
<th>P_{LM}</th>
<th>P_{SCC}</th>
<th>R_{LM}</th>
<th>R_{SCC}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>0.84</td>
<td>0.83</td>
<td>0.76</td>
<td>0.78</td>
<td>0.88</td>
<td>0.81</td>
</tr>
<tr>
<td>jFace</td>
<td>0.90</td>
<td>0.89</td>
<td>0.83</td>
<td>0.85</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td>JDT Debug</td>
<td>0.85</td>
<td>0.95</td>
<td>0.79</td>
<td>0.85</td>
<td>0.71</td>
<td>0.91</td>
</tr>
<tr>
<td>Resource</td>
<td>0.87</td>
<td>0.93</td>
<td>0.75</td>
<td>0.85</td>
<td>0.83</td>
<td>0.93</td>
</tr>
<tr>
<td>Runtime</td>
<td>0.83</td>
<td>0.91</td>
<td>0.71</td>
<td>0.85</td>
<td>0.89</td>
<td>0.83</td>
</tr>
<tr>
<td>Team Core</td>
<td>0.68</td>
<td>0.97</td>
<td>0.64</td>
<td>0.73</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>CVS Core</td>
<td>0.80</td>
<td>0.90</td>
<td>0.78</td>
<td>0.89</td>
<td>0.78</td>
<td>0.83</td>
</tr>
<tr>
<td>Debug Core</td>
<td>0.86</td>
<td>0.94</td>
<td>0.68</td>
<td>0.82</td>
<td>0.92</td>
<td>0.91</td>
</tr>
<tr>
<td>jFace Text</td>
<td>0.87</td>
<td>0.87</td>
<td>0.7</td>
<td>0.67</td>
<td>0.87</td>
<td>0.86</td>
</tr>
<tr>
<td>Update Core</td>
<td>0.78</td>
<td>0.85</td>
<td>0.77</td>
<td>0.86</td>
<td>0.85</td>
<td>0.86</td>
</tr>
<tr>
<td>JDT Debug UI</td>
<td>0.85</td>
<td>0.93</td>
<td>0.64</td>
<td>0.76</td>
<td>0.87</td>
<td>0.91</td>
</tr>
<tr>
<td>Help</td>
<td>0.75</td>
<td>0.85</td>
<td>0.75</td>
<td>0.63</td>
<td>0.69</td>
<td>0.63</td>
</tr>
<tr>
<td>JDT Core</td>
<td>0.86</td>
<td>0.87</td>
<td>0.76</td>
<td>0.82</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>OSCG</td>
<td>0.88</td>
<td>0.88</td>
<td>0.8</td>
<td>0.87</td>
<td>0.86</td>
<td>0.81</td>
</tr>
<tr>
<td>Median</td>
<td>0.85</td>
<td>0.90</td>
<td>0.75</td>
<td>0.85</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>Overall</td>
<td>0.85</td>
<td>0.89</td>
<td>0.72</td>
<td>0.74</td>
<td>0.86</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 7: AUC of E2 using different classifiers with the SCC categories as predictors for bug-prone and not-bug-prone files (AUC of the best performing classifier per project is printed in bold).
(see Table 3) may compromise the validity of the resulting model [8]. To avoid this problem, we applied principal component analysis (PCA) based on the covariance matrix and a variance threshold of 0.95. PCA extracted one component which has been used to perform the logistic regression.

The results in Table 7 show median AUC values of approximately 0.8 and higher, which indicates that all selected classifiers obtain models with adequate performance. Furthermore, we can observe that LibSVM is the best classifier for 8 projects. BNet obtains similarly good results: According to the AUC values it is the top classifier for 6 projects and has together with LibSVM the highest AUC value when learning from the entire dataset. Not surprisingly, logistic regression also yields a good performance with a high median AUC of 0.89 which is similar to the result in E1 (the input from PCA accounts for more than 0.95 of the SCC in our dataset). RFor and NN—though not the best—are still good classifiers and among the best for some projects. They fall slightly back because of their performance on the overall dataset. The decision tree methods J48 and eCHAID show lower performance compared to the other classifiers. None of them performs best for one project. Furthermore, eCHAID has the lowest median for AUC and performs the worst on the entire dataset.

Next, we compared the results of both experiments to find out whether including the information about the SCC category helps to improve the performance of prediction models. We compared the AUC values from LibSVM (the best performing classifier in E2) with the AUC values from the logistic regression in E1 using the Wilcoxon Test. The test was not significant, therefore we can conclude that the inclusion of the SCC category does not lead to better performing prediction models.

For the discussion of the performance differences between several classifiers we used a Related Samples Friedman Test and an adjusted α level for the post-hoc comparison of each classifier pair. The test was significant at α = 0.05. This means that there is a statistically significant difference between the mean ranks of the AUC values. However, a look at the pairwise tests revealed that the significance is mainly due to the low performance of eCHAID and to some extent due to J48. The differences between the other pairs that did not involve a decision tree method were not significant. These results confirm the experience drawn in prior work: (1) There is a relatively good performance of more complex classifiers in our experiments, e.g., LibSVM or RFor. But their performance does not differ statistically significant in most cases [20]; (2) the good performance of Bayesian methods [22]; and (3) in particular the comparably good predicting power of SVM for Eclipse data [32].

Based on the AUC values in Table 6 and Table 7 we conclude that SCC (E1) as well as their categories (E2) are good predictors for bug-prone and not bug-prone files. SCC outperformed the prediction models built with LM, therefore we accepted H2.

3.5 Predicting the Number of Bugs

In this section, we investigate H3—SCC is a better predictor for the number of bugs in Eclipse source files than LM.

The most common technique to solve this kind of prediction problem is linear regression. In its simplest case, the relation between bugs and source code changes is modeled as the best fitting straight line, i.e., a linear relationship is established [8]. In [4], Bernstein et al. stated that using the nonlinear MP5 regression is more adequate for this kind of data and yields better results when predicting the number of bugs compared to linear regression.

For nonlinear regression analysis, we first need to determine what type of nonlinear function, such as a polynomial, cubic, or exponential, describes the relationship between the dependent and independent variables. Figure 2 shows the scatterplot of the CVS Core project on file level. The plot shape is representative for all the Eclipse projects in our dataset. One can see that a straight line does indeed not fully capture the characteristic of the relationship as stated in [4]. The curve that fits best exhibits a steep slope in the beginning and then flattens out to some extent as SCC moves towards large values. This can be interpreted as: When a file already has been subject to a large amount of changes, each additional change is probably less and less important with respect to an increase in Bugs. This is similar to the sigmoidal s-shaped function that underlies the logistic regression we used in Section 3.4, and that models a saturation effect in terms of probabilities.1

An appropriate model for such data as in Figure 2 is the asymptotic model described by the function (see [30]):

\[
f(x) = b_1 + b_2 \times e^{b_3 \times \text{SCC}} \quad \text{with} \quad b_1 > 0, b_2 < 0, \text{and} \quad b_3 < 0
\]

We used this function to compute the nonlinear regression once with LM and once with SCC as independent variables and Bugs as the dependent variable.

Table 8 lists the resulting R² values of validating the models with 10 fold cross validation. R² is the coefficient of determination that shows how much of the variance in the dataset is explained by the obtained predicting model. Overall denotes the performance of the model that resulted when merging all files into one dataset. With a median R²_{SCC} of 0.79 the models using SCC exhibit good explanatory power across all projects. Four projects even exhibit an R²_{SCC} of 0.85 or higher. These models explain a large amount of the variance in their respective dataset. There are three projects in our dataset where nonlinear regression has lower explanatory power meaning an R²_{SCC} < 0.7: In Update Core not even half of the variance is explained by the model; in JDT Debug and Help around two third of the variance is explained. An average Spearman correlation of 0.77 indicates the sensitivity

1Logistic regression itself is a nonlinear regression when the dependent variable is non-numerical, e.g., dichotomous.
of the models, \textit{i.e.}, an accompanied increase/decrease of the actual and the predicted number of bugs.

With an average $R^2_{LM}$ of 0.7, LM has less explanatory power compared to SCC using an asymptotic model. Except for the case of JDT Debug UI having equal values, LM performs lower than SCC for all projects including Overall. The Related Samples Wilcoxon Signed-Ranks Test on the $R^2$ values of LM and SCC in Table 8 was significant, denoting that the observed differences in our dataset are significant.

To assess the validity of a regression model one must pay attention to the distribution of the error terms. Figure 3 shows two examples of fit plots with normalized residuals (y-axis) and predicted values (x-axis) of our dataset: The plot of the regression model of the Overall dataset on the left side and the one of Debug Core having the highest $R^2_{SCC}$ value on the right side. On the left side, one can spot a "funnel" which is one of the "archetypes" of residual plots and indicates that the constance-variance assumption may be violated, \textit{i.e.}, the variability of the residuals is larger for larger predicted values of SCC [19]. This is an example of a model that shows an adequate performance, \textit{i.e.}, $R^2_{SCC}$ of 0.72, but where the validity is questionable. On the right side, there is a first sign of the funnel pattern but it is not as evident as on the left side. The lower part of Figure 3 shows the corresponding histogram charts of the residuals. They are normally distributed with a mean of 0.

Therefore, we accept $H_3$–SCC (using asymptotic nonlinear regression) achieves better performance when predicting the number of bugs within files than LM. However one must be careful to investigate wether the models violate the assumptions of the general regression model. We analyzed all residual plots of our dataset and found that the constance-variance assumption may be generally problematic, in particular when analyzing software measures and open source systems that show highly skewed distributions. The other two assumptions concerning the error terms, \textit{i.e.}, zero mean and independence, are not violated. When using regression strictly for descriptive and prediction purposes only, as it is the case for our experiments, these assumptions are less important, since the regression will still result in an unbiased estimate between the dependent and independent variable [19]. However, when inference based on the obtained regression models is made, \textit{e.g.}, conclusions about the slope coefficients) or the significance of the entire model itself, the assumptions must be verified.

### 3.6 Summary of Results

The results of our empirical study can be summarized as follows:

**SCC correlates strongly with Bugs**. With an average Spearman rank correlation of 0.77, SCC has a strong correlation with the number of bugs in our dataset. Statistical tests indicated that the correlation of SCC and Bugs is significantly higher than between LM and Bugs (accepted $H_1$).

**SCC categories correlate differently with Bugs**. Except for $cDecl$ all SCC categories defined in Section 3.1 correlate substantially with Bugs. A Friedman Test revealed that the categories have significantly different correlations. Post-hoc comparisons confirmed that the difference is mainly because of two groups of categories: (1) $stmt$, $func$, and $mDecl$, and (2) $else$, $cond$, $oState$, and $cDecl$. Within these groups the post-hoc tests were not significant.

**SCC is a strong predictor for classifying source files into bug-prone and not bug-prone**. Models built with logistic regression and SCC as predictor rank bug-prone files higher than not bug-prone with an average probability of 90%. They have a significant better performance in terms of AUC than logistic regression models built with LM as a predictor (accepted $H_2$). In a series of experiments with different classifiers using SCC categories as independent variables, LibSVM yielded the best performance—it was the best classifier for more than half of the projects. LibSVM was closely followed by BNet, RFor, NBayes, and NN. Decision tree learners resulted in a significantly lower performance. Furthermore, using categories, \textit{e.g.}, $func$, rather than the total number of SCC did not yield better performance.
SCC is a strong predictor for the number of bugs in source files. Asymptotic nonlinear regression using SCC showed high explanatory power with a median $R^2_{SCC}$ of 0.79 and significantly outperforms the regression models computed with LM (accepted H3).

4. DISCUSSION

The results of our study showed that the use of SCC improves bug prediction models significantly. The models computed with SCC outperformed the models computed with LM (i.e., code churn). As a result, our models based on SCC can help allocating resources more efficiently to bug-prone parts of a system, i.e., those parts where most of the defects are expected.

The gain in performance comes with the additional effort that is needed to extract the fine-grained source code changes from the project history. This is, however, not an issue when tools, such as CHANGE DISTILLER, are available that perform this extraction fully automatically (e.g., during nightly builds).

The comparison of different classifiers confirmed the results of prior work and showed the strength of advanced machine learning techniques, in particular LibSVM. The importance of the differences in performance should not be overestimated. For instance, the differences between LibSVM, BNNet, RF, NB, and NN were not significant in terms of AUC; only the decision tree methods J48 and eCHAID performed significantly lower. As a consequence one might consider other criteria, such as readability and interpretability of the resulting models when choosing an appropriate classifier. In [9], Fenton and Neil argue that multivariate methods often result in models that are difficult to understand. For instance, PCA used for logistic regression in E2 produces components that are delicate to explain causally. Similarly, SVM and NN often lack explanatory power, i.e., it is difficult for end users to extract concrete rules as their internal mechanisms are complex to interpret, and the insights into their learning process and decisions are limited [28, 34]. In contrast, decision tree learners produce rules that are easier to extract [18].

Despite the fact that some classification techniques outperform others, an analysis of the distribution of the values and a correlation analysis need to be performed first for this type of experiments. With a value of 0.48 the Help project showed the lowest correlation between Bugs and SCC (see Table 5). Consequently, all selected classifiers resulted in AUC values below 0.7 in that particular case (see Table 7). Similar results were obtained for Team Core, which showed almost no correlation (0.15) between LM and Bugs and consequently low values for AUC (see Table 6). This confirms and strengthens the results from prior work, e.g., [26, 39], that an initial correlation analysis can not only reveal multicollinearity in the dataset but also give a first idea of the strength of the relationship between variables and what their predictive power is.

In Section 3.5, we performed a study to predict the number of bugs in files using regression analysis with SCC. The distributions of our dataset and the highly skewed and non-normal distribution of software properties [3] suggested that linear regression is not appropriate for such data. We recommend to use nonlinear regression that better represents the data. The experiments showed that an asymptotic model with a median $R^2_{SCC}$ of 0.79 has high explanatory power. The prediction models of 7 projects showed an $R^2_{SCC}$ of 0.8 or higher, 4 had values of 0.85 or higher. An analysis of the residuals indicates that the constance-variance assumption is violated in some cases. Therefore, such models must not be used for inference purposes because the results of (inference) tests are possibly biased. Since we use the models mainly in a descriptive manner, this assumption is of less concern.

5. THREATS TO VALIDITY

From an external validity point of view this work is possibly biased by our sole focus on Eclipse projects. Although we collected data from 15 different projects that vary in terms of size, source code changes, and their respective function, they are all part of the larger Eclipse platform. This might question the generalizability of the results and findings for other software systems. In fact, every conclusion based on empirical work is threatened by the bias of the dataset it was drawn from [22]. Especially in software engineering where the development process of a system depends on a large number of factors that potentially vary widely across different systems and domains, the issue of sampling bias may be more prominent [2]. Nevertheless, Eclipse is a representative case study that emerged to a standard IDE since its first release in 2001. It has been studied extensively before, and we can build upon the valuable findings of prior work, e.g., [24, 29, 32, 39]. Therefore, our study contributes to an existing body of knowledge, strengthen existing hypothesis, and presents new results.

Threats to internal validity arise from two measurement issues: (1) We counted the number of bugs and established the link between bug data and source files by searching references to bug reports in the log messages of the versioning system (Section 2). We rely on the fact that bug fixes are consistent and recorded manually. Bird et al. reported on evidence about a systematic bias in bug datasets [5]. (2) When comparing the ASTs of two revisions, CHANGE DISTILLER occasionally extracts a nonoptimal set of changes, i.e., more changes than actually required for AST transformation. However, the transformation itself between two AST versions is always correct. The accuracy of the change extracting algorithm was evaluated using a benchmark in [13].

6. RELATED WORK

Since software defects are an important cost factor and development teams often operate with limited time and budget constraints, building bug prediction models is an active research field. There are roughly three main factors upon which prediction models are based: Product and process measures and organizational aspects—or a combination of them. Product measures are directly computed on the source code. In particular, complexity and size metrics have been investigated to build prediction models [1, 9, 22]. The rationale is that larger and more complex parts of a system contain more defects. Several approaches used source code dependency information. Findings from [37] showed that the position of a binary within the static dependency graph of Windows Server 2003 correlates with the number of post-release failures. Nguyen et al. replicated this study on the Eclipse project [29]. In [32], the import relationship of Eclipse files and packages achieves good predicting power. Similar to our results SVM performed the best. The good predictive power of advanced classifiers, e.g., SVM, Random Forest, and Neural Networks, was confirmed by Lessman et al. [20]. They
compared the performance for defect prediction of different learning algorithms using product measures of the NASA dataset. Despite the good performance of some classifiers compared to others, no significant difference could be detected.

Process measures are often obtained from software repositories. SCC, as used in this study, falls under this category. Among the first to study the relation between code churn defined as LM and Bugs was [17]. Work carried out in [25] explored the extent to which the use of relative code churn measures, e.g., LM weighted by total lines of code, outperformed absolute measures when predicting defect density: In Windows Server 2003, absolute churn measures showed a lower performance compared to relative ones. The results of several studies showed that process measures might be better defect predictors than product measures: Graves et al. found out that the number of changes and the age of a module yield better prediction models than product measures [15]. A comparative study showed that, especially in Eclipse, process measures outperformed product measures [24]. In [18], the J48 decision tree with a combination of product and process measures were used to predict defect density of Mozilla releases. The results showed that process measures are good predictors. The extent to which measuring within different time frames improves bug prediction was investigated in [4]. Consistently with our experiments, prior work validated the usefulness of nonlinear models for building prediction models based on process measures [4, 15]. A study on Windows Vista showed that the number of consecutive changes rather than the number of single changes have high predictive power [27].

Organizational measures describe the management circumstances that influence the development of software. Bird et al. compared the failure differences between components of Windows Vista that were developed in a distributed way and those that were developed at collocated sites [6]. Contrary to common wisdom they stated that geographical differences had little or no effect on failures. A strong organizational indicator of software quality in Windows Vista is developer contribution [31]: The number of developers working on a binary positively correlates with post-release failures.

Recent work focused the discussion on prediction models themselves. A critical review about the current state of the art regarding defect prediction models is given in [9]. For instance, the authors mention that current prediction models suffer from problems in statistical methods and data quality. Following the results presented in [22] a discussion emerged about the practical usefulness of defect prediction models [21, 36]. Prediction models require a sufficient amount of initial training data. Often, such data is not available beforehand. Therefore, Zimmerman et al. raised the importance of exploring the cross-project prediction ability of models, i.e., applying the models to data of a project other than it was obtained from [38]. Their results of reapplying models trained on data from different Microsoft products and several open source projects among each other showed that cross-project prediction is a serious challenge.

7. CONCLUSION AND FUTURE WORK

In this paper, we empirically analyzed the relationship between fine-grained source code changes (SCC) and the number of bugs in source files (Bugs) using data from the Eclipse platform. Based on an initial correlation analysis, we computed a set of prediction models using several machine learning methods. The results of our study are:

- SCC shows a significantly stronger correlation with the number of bugs than code churn based on lines modified (LM) (accepted H1).
- Classification models using SCC rank bug-prone files higher than not bug-prone ones with an average probability of 90%. This is an improvement compared to models computed with LM (accepted H2).
- Although advanced learning methods performed better, we could not always observe a significant difference between them.
- Nonlinear asymptotic regression using SCC obtained models to predict the number of bugs with a median $R^2$ of 0.79 which is an improvement over models computed with LM (accepted H3).

Our results clearly show the good performance of SCC and the improvements over LM for bug prediction. This can help allocating maintenance and testing resources to bug-prone parts of a software system.

Currently, our dataset is solely Eclipse focused. Therefore, conclusions made in this work can be biased by characteristics of the development process that are specific and unique to Eclipse. To address this issue replications of our study with other projects are required [2]. Regarding our prediction models, we plan to use other cut points than the median, e.g., the third quartile, to investigate how this affects their performance. Furthermore, including information about the categories of change types did not result in better prediction performance although some categories showed a stronger correlation with bugs than others. We plan to investigate the relationship between bugs and categories of change types more in depth, e.g., which change types are used to fix bugs. The choice of an asymptotic regression model was based on the analysis of the scatterplots. However, more complex or segmented regression models exist that we plan to explore.

8. REFERENCES


