Predicting Defect Densities in Source Code Files with Decision Tree Learners

Patrick Knab, Martin Pinzger, Abraham Bernstein
Department of Informatics
University of Zurich, Switzerland
{knab,pinzger,bernstein}@ifi.unizh.ch

ABSTRACT

With the advent of open source software repositories the data available for defect prediction in source files increased tremendously. Although traditional statistics turned out to derive reasonable results the sheer amount of data and the problem context of defect prediction demand sophisticated analysis such as provided by current data mining and machine learning techniques.

In this work we focus on defect density prediction and present an approach that applies a decision tree learner on evolution data extracted from the Mozilla open source web browser project. The evolution data includes different source code, modification, and defect measures computed from seven recent Mozilla releases. Among the modification measures we also take into account the change coupling, a measure for the number of change-dependencies between source files. The main reason for choosing decision tree learners, instead of for example neural nets, was the goal of finding underlying rules which can be easily interpreted by humans. To find these rules, we set up a number of experiments to test common hypotheses regarding defects in software entities. Our experiments showed, that a simple tree learner can produce good results with various sets of input data.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

Data Mining, Defect Prediction, Decision Tree Learner

General Terms

Measurement, Management, Reliability

1. INTRODUCTION

A successful software project manager knows how to direct his resources into the areas with the highest impact on the bottom line. Regarding the quality of a software system, the areas with great impact are the parts of the code base with the highest defect density, or even better, with the most future problem reports. Problem reports obtainable from issue tracking systems (*e.g.,* Bugzilla) can be used to assess the perceived system quality with respect to defect rate and density. The objective of such an assessment is to identify the code parts (*i.e.,* software modules) with the highest defect density. Improving them will allow the software developers to reduce the number of problem reports after delivery of a new system or an update.

Our long term goal is to provide software project teams with tools allowing a manager to invest resources proactively (rather than reactively) to improve software quality before delivery. In this paper we address the issue of predicting defect densities in source code files. We present an approach that applies decision tree learners to source code, modification, and defect measures of seven recent source code releases of Mozilla's content and layout modules. Using this data mining technique we conduct a series of experiments addressing the following hypotheses:

1. *Hyp 1*: We can derive defect-density from source code metrics for one release.
   This hypothese covers two sub hypotheses concerned with code quality assessment.
   - *Hyp 1a*: Large source code files have a higher number of defects than small files.
     This is a popular premiss with the underlying assumption that large files are complex, hard to understand and therefore more susceptible to defects. However, there is little to gain here. Even if we assume a balanced distribution of defects, larger files trivially have more defects. More interesting is the defect-density, *i.e.,* number of problem reports per line of code. Which gives us:
   - *Hyp 1b*: Larger files have a higher defect-density.
2. *Hyp 2*: We can predict future defect-density.
   This is the holy grail of software project management. If we can predict the files which will have the highest defect rate in a future release, this would certainly help with ressource allocation in a project.
3. *Hyp 3*: We can identify the factors leading to high defect-density.
   Knowing locations with highest defect density the next step is concerned with gaining insights into the reasons that lead to defects. These insights allow software developers to proactively improve the system and reduce the number of post-release defects.
4. *Hyp 4a*: Change couplings contain information about defect-density in source files of a single release.

MSR'06, May 22–23, 2006, Shanghai, China.

Change coupling has shown to provide valuable information for analyzing change impact and propagation [13, 15]. In this work we take into account the measure of the change coupling strength and test its defect density predictive capability in a single release and:

5. *Hyp 4b*: Change couplings contain predictive information about the number of defects in future releases.

Our experiments showed, that a simple tree learner can produce good results with various sets of input data. We found that common rules of thumb, like lines of code are of little value for predicting defect densities. On the other side, "yesterday's weather" [6], that is, number of bug reports in the past, was one of the best predictors for the future number of bug reports. We also saw, that when we removed various attributes from the input data, the learning algorithm was able to keep its performance, by selecting other, often surprising, attributes.

The remainder of the paper is organized as follows: Related work is presented in Section 2. Section 3 describes the data we used for our experiments. The experiments including a discussion of the results are presented in Section 4. Section 5 draws the conclusions and indicates areas of future work.

2. RELATED WORK

The need for better guidance in software projects to proactively improve software quality led to several related approaches. In this work we concentrate on predicting defect density as well as the number of defects.

A number of approaches concentrated on using code churn measures (*i.e.,* amount of code changes taking place within a software unit over time) for fault and defect density prediction. For instance, Khoshgoftaar *et al.* [9] investigated the identification of fault prone modules in a large software system for telecommunications. Software modules are defined as fault-prone when the debug churn measure (amount of lines of code added or changed for fixing bugs) exceeds a given threshold. They applied discriminant analysis to identify the fault-prone modules based on sixteen static product metrics and the debug churn measure.

Most recently, Nagappan and Ball [12] presented a technique for early prediction of system defect density based on code churn measures. Their main hypothesis is that code that changes many times pre-release will likely have more post-release defects than code that changes less over the same period of time. Addressing this hypothesis the authors showed in an experiment that their relative (normalized) code churn measures are good predictors for defect density while absolute code churn measures are not. In this paper we also address the issue of total and relative metric values but concentrate on different source code metrics of several releases instead of code churn measures solely. Further we apply machine learning techniques for our defect density prediction instead of using statistical regression models.

Munson *et al.* [11] used discriminant analysis and focused on the relationship between program complexity measures and program faults which are found during development. Besides lines of code and related metrics *e.g.,* character count, they use Halstead's program length, Jensen's estimator of program length, McCabe's cyclomatic complexity and Belady's bandwidth metric. Due to the high collinear relationship of these metrics, they mapped them with a principle-components procedure in two distinct, orthogonal complexity domains. They found that, although the detection of modules with high potential for faults worked well, the produced models were of limited value. In our work we use different metrics, especially various coupling metrics (*e.g.,* fan in and fan out). Additionally we build our model from multiple releases with decision tree learners.

Fenton *et al.* [4] tested a range of basic software engineering hypotheses and found that a small number of modules contain most of the faults discovered in pre-release testing and that a very small number of modules contain most of the faults discovered in operation. However, they found, that in neither case it could be explained by the size or complexity of the modules. They distinguished between pre- and post-release fault discoveries, whereas we concentrate on bug reports, which are mostly post-release. We can confirm the findings of Fenton *et al.* regarding the relevance of module size (in our case file size), and their observation concerning the distribution of faults discovered in operation.

In addition to the complexity measures a number of objectoriented software metrics have been developed such as the ones from Chidamber and Kemerer [3]. As with the complexity measures, the results and opinions of the various investigations are different. An early investigation of these metrics comes from Basili *et al.* [1]. They have defined a number of hypotheses regarding the fault-proneness of a class. To validate these hypotheses they conducted a student's project in which the students had to collect data about the faults found in a program. Based on this data they used univariate logistic regression to evaluate the relationship of each of the metrics in isolation and fault-proneness and multivariate logistic regression to evaluate the predictive capability. The results have shown that all but one of these metrics are useful predictors of fault-proneness.

Ostrand *et al.* [2] used a negative binary regression model to predict the location and number of faults in large software systems. The variables for the regression model were selected using the characteristics they identified as being associated with high fault rates. They also found, that a simplified model only based on file size was only marginally less accurate. We can support the finding that lines of code is a good measure for number of faults, from our research. However, this fact is of little help in the management of the development process. To reduce the overall number of faults, we have to reduce the fault density. The focus of our work is more on the understanding of the factors that lead to faults than the actual fault prediction.

Graves *et al.* [7] developed several statistical models to evaluate which characteristics of a module's change history were likely to indicate that it would see large numbers of faults generated as it is continued to be developed. Their best model, a weighted time damp model, predicted fault potential using a sum of contributions from all the changes to the module in its past. Their best generalized linear model used numbers of changes to the module in the past together with a measure of the module's age. They found, that the number of deltas, *i.e.,* the number of changes was a successful predictor of faults, which is also indicated by our experiments. They also found, that change coupling is not a powerful predictor of faults, which our results also support. By using decision trees we use all available measures to build a model including past modification reports, change couplings and various source code metrics.

Hassan and Holt [8] presented heuristics derived from caching mechanisms to find the ten most fault susceptible subsystems which they tested on several big open source projects. Their heuristics are based on the subsystems that were most recently modified, most frequently fixed, and most recently fixed. Although we did not distinguish between repairing modifications and general modifications, most of the information is also contained in our metrics.

Finally, Mohagheghi *et al.* [10] concentrated on the influence of code reuse on defect-density and stability. They found that reused components have lower defect-density than not reused ones. They did not observe any significant relation between the number of

defects, and component size. They neither found a relation between defect-density and component size. Our results support the second finding, but contradict the first.

3. EXPERIMENTAL SETUP

The data for our experiments stems from seven releases of the content and layout modules of the Mozilla open source project. [1] The modules are: DOM, NewLayoutEngine, XPToolkit, NewHTML-StyleSystem, MathML, XML, and XSLT. For more information on these modules we refer the reader to the module owners web-site [2] of the Mozilla project. The selected releases and their release dates are listed in Table 1.

#	Release	Date
1	0.92	June, 2001
2	0.97	December, 2001
3	1.0	June, 2002
4	1.3a	December, 2002
5	1.4	June, 2003
6	1.6	January, 2004
7	1.7	June, 2004

Table 1: Selected Mozilla releases.

In release 1.7 the seven content and layout modules comprise around 1.300 C/C++ source and header files with a total of around 560,000 lines of code. From this set of files we selected 366 out of 504 *.cpp files. We skipped 138 files because they did not show a complete history as is needed for our experiments (*i.e.,* they were added/removed during this time period). We also skipped the header files (817 *.h files) because they are naturally connected with the corresponding implementation files. So, there is nothing to gain with respect to analyzing the change coupling and predicting the defect density of these source files.

For this set of *.cpp source files per release we computed the source code, modification, and defect report metrics as listed in Table 2. For the source code metrics we parsed each source code release using the Imagix-4D C/C++ analysis tool. [3] The modification and defect report metrics were retrieved from the release history database that we extracted from Mozilla's CVS and Bugzilla repositories as has been presented in our previous work with this project [5].

The first three source code metrics listed in Table 2 quantify the size of a *.cpp file according the lines of code (linesOfCode), the number of defined global and local variables (nrVars), and the number of implemented functions/methods (nrFuncs). The following four source code metrics quantify the strength of the static coupling of a *.cpp file with other *.cpp files. For our experiments we consider incoming (incomingCallRels) and outgoing function calls (outgoingCallRels) as well as incoming (incomingVarAccessRels) and outgoing variable accesses (outgoingVarAccessRels).

The remaining metrics are retrieved from the release history database and computed for the time from the begin of the Mozilla project to the selected release dates. They denote the number of checkins of a *.cpp file (nrMRs), the number of times a file was checked in together with other files (sharedMRs), and the number of reported problems (nrPRs). For the latter metric we further detail the measures into additional categories denoting the different severity levels of reported problems. These levels range from problem reports that are marked as trivial to system critical problem reports (*i.e.,*, system crashes, loss of data). They allow us a more detailed classification of the defects in source files.

[1] http://www.mozilla.org/

[2] http://www.mozilla.org/owners.html

[3] http://www.imagix.com

Name	Description
linesOfCode	Lines of code
nrVars	Number of variables
nrFuncs	Number of functions
incomingCallRels	Number of incoming calls
outgoingCallRels	Number of outgoing calls
incomingVarAccessRels	Number of incoming variable accesses
outgoingVarAccessRels	Number of outgoing variable accesses
nrMRs	Number of modification reports
sharedMRs	Number of shared modification reports
nrPRs	Number of problem reports
nrPRsNormal	nrPRs with severity = normal
nrPRsTrivial	nrPRs with severity = trivial
nrPRsMinor	nrPRs with severity = minor
nrPRsMajor	nrPRs with severity = major
nrPRsCritical	nrPRs with severity = critical
nrPRsBlocker	nrPRs with severity = blocker

Table 2: Base metrics computed for a C/C++ file.

The shared modification reports metric (sharedMRs) represents the number of times a file has been checked into the CVS repository together with other files. The reason for adding this metric is that the defect density of a file is higher when modifications (*e.g.,*, bug fixes) are spread over several files instead of being local to one source file. This metric has been used several times in recent investigations to assess the quality of software systems and their evolution (see for example [13, 15]). In this paper we test its defect density predictive capability (see Hyp 4a and Hyp 4b).

The metrics listed above are all computed for each selected release. For predicting the defect density of files we further added trend and normalized values of these metrics. Trends are denoted by the deltas of metric values between two subsequent releases. For instance, the number of functions added/removed or the number of critical problem reports reported from one release to the next. Total as well as delta values are normalized with the size of a file expressed in lines of code (linesOfCode). Such a normalization is a key factor for predicting the defect density namely the number of new defects per line of code.

Total and delta values as well as their normalized values form the input to the experiments presented in the following sections. Regarding the metric names used in the experiments we prefix each metric name with the kind of value: total metrics with "static_"; normalized metrics with "norm_"; and trend metrics with "delta_". Furthermore, the number indicating the release (see Table 1) is added to each metric name. For instance, delta_nrMRs_4 denotes the number of modification reports added from release 1.0 to release 1.3a.

4. EXPERIMENTS

Before we go into our data mining experiments we conducted a number of descriptive statistics analysis with the selected Mozilla releases. Here we present an excerpt of the results we obtained for the Mozilla release 1.0. Similar observations apply to the other Mozilla releases. Concerning Hyp 1a and Hyp 1b the scatter plot in Figure 1 shows that number of problem reports in release 1.0 display a strong linear correlation with lines of code. So big files do not have a higher problem reports to lines of code ratio which shows us that at least for Mozilla the popular belief that big files are

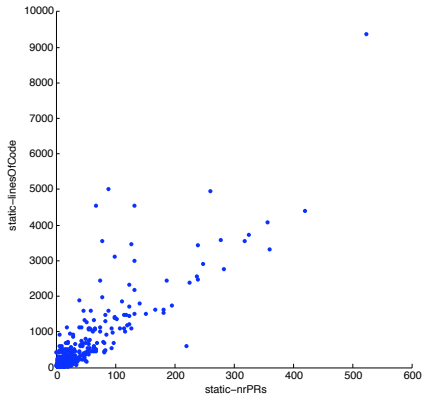

Figure 1: lines of code vs number of problem reports

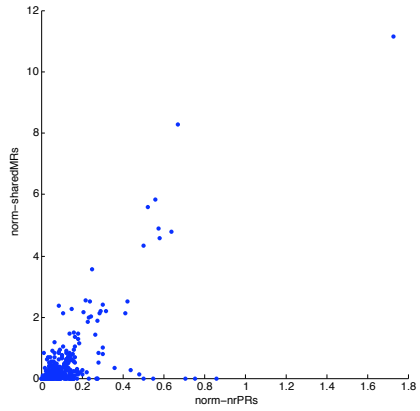

Figure 2: normalized shared modification reports versus normalized problem reports

trouble files does not hold. The downside of this observation is that lines of code does not make for a simple indicator to detect problem files.

Our use of lines of code per file instead of, for example, lines of code per class, results from the fact that most of our metrics like shared modification reports or problem reports, are only calculated for files. Lines of code per class might be of some significance when assessing defect-density, however, this is, based on the research of Fenton and Ohlsson [4], not the case. In the data mining experiments we will further elaborate on the issue whether lines of code has any predictive value.

To test Hyp 4a we analyze the correlation between the normalized values of shared modification reports and problem reports. The scatter plot for Mozilla release 1.0 is shown in Figure 2. The correlation coefficient is 0.7234, which, in combination with the graphic, shows a strong linear correlation between the two values. However, what value, the normalized shared modification reports metric presents for the prediction of future number of defects, remains to be seen.

The process for all data mining experiments is as follows: We export the selected data to an arff file (*i.e.,* a text based data file readable by the WEKA [14] explorer), which is then loaded into the WEKA explorer. We then run the five bins equal frequency discretizer over our data to get the input for our classifier. The use of equal frequency distribution in the discretizer means that the prior probability for an instance falling into a given class is twenty percent. The classifier is the J48 tree learning algorithm provided by the WEKA tool. The accuracy is calculated with ten-fold cross validation.

Exp 1: Problem reports from non PR metrics of the same release: In the first experiment we use all available data from release four (1.3a) excluding problem report metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.) to predict the number of problem reports of release four (nrPRs_4). Figure 3 depicts the top levels of the generated decision tree. We can see, that the attribute with the most information concerning the number of problem reports is the number of modification reports, hence it appears at the root. Attributes on the second level are: added number of modification reports since release 3, shared modification reports, and lines of code. We got

```
Correctly Classified Instances   227   (62.0219 %)
Incorrectly Classified Instances 139   (37.9781 %)
```

which is good, given the prior probability of 0.2. Looking at the confusion matrix

```
 a  b  c  d  e   <-- classified as
 60  9  3  1  0 |  a = '(-inf-7.5]'
 12 40 22  2  0 |  b = '(7.5-15.5]'
  6 22 25 18  0 |  c = '(15.5-25.5]'
  1  3 16 42 11 |  d = '(25.5-62.5]'
  0  0  0 13 60 |  e = '(62.5-inf)'
```

we see the detailed performance for our five classes selected by the discretizer. The top row of the confusion matrix shows the labels of the predicted classes. On the right are the labels and the corresponding intervals of the actual classes. Each cell of the matrix denotes the number of instances (source files) classified as a, b, c, d, or e. The matrix diagonal contains the exact matches. For instance, the numbers in the bottom row state that 60 instances which are of the actual class e (*i.e.,* source files with more than 62.5 problem reports), where classified correctly. 13 instances where wrongly classified as d, none as c, b, or a.

Taking into account only the worst twenty percent, the algorithm gets 82 percent right, and the other 18 percent are put into the second worst class. From a management's point of view, this presents a valuable result. If the manager concentrates his ressources on the files which were classified as e or d (*i.e.,* the worst and second worst class), he would have covered 100 percent of the worst files (*i.e.,* the files with the highest number of defects).

The connection between the number of modification reports and problem reports is not that surprising. If there are many bugs, one has to fix them, which generates modification reports. So what happens if we take the modification reports away from our learning algorithm?

Exp 2: Problem reports from non PR metrics of the same release without MR data: We do the same experiment as above using the available metrics from release four (1.3a) excluding modification report data (*i.e.,* nrMRs, sharedMRs) and problem report metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.). With this data we predict number of problem reports per line of code (norm_nrPRs_4) for release four (1.3a). Normalized problem reports are better suited to assess the badness of a file, because big files with a low defect-density are rated better than small files stuffed with bugs.

Results are below:

```
Correctly Classified Instances      138   (37.7049%)
Incorrectly Classified Instances    228   (62.2951%)
```

Figure 3: Top levels of decision tree resulting from Exp 1

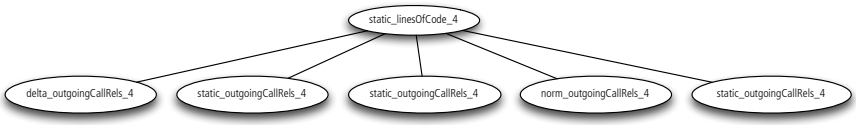

Figure 4: Top levels of decision tree resulting from Exp 2

```
 a  b  c  d  e   <-- classified as
 26 20 11  7  9 |  a = '(-inf-0.037225]'
 19 20 19  4 11 |  b = '(0.037225-0.065399]'
  9 22 22 10 10 |  c = '(0.065399-0.099327]'
  9 12  9 34 10 |  d = '(0.099327-0.143163]'
  5 15  9  8 36 |  e = '(0.143163-inf)'
```

The quality, although significantly above the prior probabilities, is much worse. Still, by looking at the tree in Figure 4, we can see that there is at least some information in the size of a file for the prediction of the number of problem reports. Although, by looking at the confusion matrix, we can see that the predictions are heavily scattered which makes them hardly useful. For assessing the importance of modification reports data we have to conduct additional experiments.

Exp 3: Normalized problem reports from non PR metrics of the same release: Here we derive the normalized number of problem reports from all, but problem report metrics. Thus repeating experiment one with normalized problem reports as the target class. To predict number of problem reports per line of code (norm_nrPRs_4) of release four (1.3a) we use all metrics from release four except PR metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.). We get:

```
Correctly Classified Instances    192  (52.459 %)
Incorrectly Classified Instances  174  (47.541 %)
```

for this experiment, which confirms the results of experiment one, but differs in at least one important way. Lines of code is not present in the top levels of the resulting tree as shown in Figure 5. In the full tree lines of code is only used in one branch on the third level. This confirms that lines of code is of marginal importance for the prediction of defect-density and lets us reject [4] Hyp 1a, and Hyp 1b.

The confusion matrix illustrates the good performance of the classifier. By looking at the diagonal we see moderate dispersion of the values. If we count near misses, the prediction, especially for class e, is excellent.

```
a  b  c  d  e   <-- classified as
 48 19  4  2  0 |  a = '(-inf-0.037225]'
 14 33 15  7  4 |  b = '(0.037225-0.065399]'
  6 16 33 13  5 |  c = '(0.065399-0.099327]'
  3  8  7 35 21 |  d = '(0.099327-0.143163]'
  1  2  7 20 43 |  e = '(0.143163-inf)'
```

[4]This is an informal rejection as we have not used any formal hypotheses testing model such as T-test.

Exp 4: Normalized problem reports from non PR metrics of the same release without sharedMR data: Here we exclude shared modification report metrics thus using all metrics except PR metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.) and shared modification report metrics (*e.g.,* sharedMRs, norm_sharedMRs) from release four, to predict the number of problem reports per line of code (norm_nrPRs_4) of release four (1.3a).

```
Correctly Classified Instances    197  (53.8251 %)
Incorrectly Classified Instances  169  (46.1749 %)
```

```
 a  b  c  d  e   <-- classified as
 49 17  3  4  0 |  a = '(-inf-0.037225]'
 14 35 14  6  4 |  b = '(0.037225-0.065399]'
  3 16 37 13  4 |  c = '(0.065399-0.099327]'
  2  7 13 29 23 |  d = '(0.099327-0.143163]'
  1  5  2 18 47 |  e = '(0.143163-inf)'
```

The error rate and the confusion matrix are almost identical to experiment three. This is a strong sign, that the number of problem reports does not depend on the amount of logical coupling a file has with its surrounding.

But, taking a closer look at Figure 6 we can see that other coupling metrics were used in the prediction of number of problem reports: added normalized outgoing call relationships and incoming call relationships.

The results of these first experiments show, that we can predict defect densities (measured by number of problem reports) with accuracies of more than 50% given a prior probability of 20%. So we can accept Hyp 1. However, lines of code is not a good predictor of defect-density so we have to reject Hyp 1a and Hyp 1b. At this point, we cannot verify Hyp 4a fully. The classifier uses mainly other modification report metrics for the prediction which indicates a low importance of shared modification reports for defect prediction.

The next set of experiments are mainly concerned with Hyp 2 and Hyp 4b.

Exp 5: Added problem reports of release 6 with data from releases 3, 4, 5: For experiment four we use all available data from releases 3, 4, and 5, *e.g.,* lines of code in release three (linesOfCode_3), added modification reports in release four (delta_nrMRs_4), the number of added problem reports with severity major per lines of code in release five (delta_norm_nrPRsMajor5) and predict the number of added problem reports in release 6 (delta_nrPRs_6).

The performance of the classifier is acceptable:

```
Correctly Classified Instances    186  (51.2397%)
```

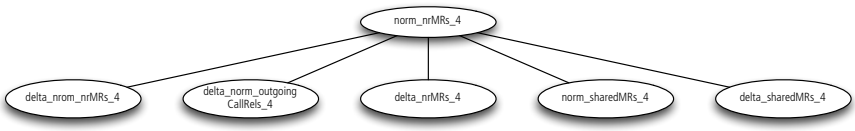

Figure 5: Top levels of decision tree resulting from Exp 3

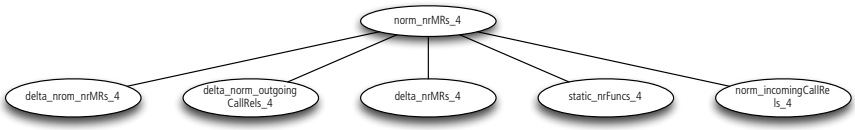

Figure 6: Top levels of decision tree resulting from Exp 4

```
Incorrectly Classified Instances  177  (48.7603%)

 a   b   c   d   e   <-- classified as
104  11  12   9   2 |   a = '(-inf-0.5]'
 26  16   2   4   0 |   b = '(0.5-1.5]'
 25   4  23  11   4 |   c = '(1.5-3.5]'
 12   3  14  12  15 |   d = '(3.5-6.5]'
  3   0   4  16  31 |   e = '(6.5-inf)'
```

The confusion matrix shows a high accuracy for class a, lower accuracy for the middle classes (b,c,d), and again a high accuracy for class e if we count near misses.

In Figure 7 we see that the top node, added problem reports with severity major from release 4, divides the data set the best regarding added problem reports. The presence of change coupling metrics only in a few lower branches shows that isolated, they are not very valuable for the prediction of the future number of defects. Which supports our finding, that there are no simple dependencies between defect-density and other metrics.

Exp 6: Added problem reports of release 7 with data from releases 3, 4, 5, 6: This experiment is a repetition of Exp 5 but predicting for release seven (delta_nrPRs_7) using input data from releases three through six. As we can see, from the output below, the performance is better and, more interesting, the top node has changed to something, at least for us, surprising. In Figure 8 the top node of the tree is static_nrFuncs_6. At the second level, however, mostly problem report metrics from earlier releases are used.

```
Correctly Classified Instances     215  (59.2287%)
Incorrectly Classified Instances   148  (40.7713%)

  a   b   c   d   e   <-- classified as
145  17   2   1   2 |   a = '(-inf-0.5]'
 35  14   7   5   5 |   b = '(0.5-1.5]'
  7   8   6   6   2 |   c = '(1.5-2.5]'
  8   7   7  18  12 |   d = '(2.5-4.5]'
  3   4   2   8  32 |   e = '(4.5-inf)'
```

Conducting the same experiment with normalized added problem reports as target attribute, the performance degrades to:

```
Correctly Classified Instances      162  (44.6281%)
Incorrectly Classified Instances    201  (55.3719%)
```

This result supports our assumption that number of functions is used as a measure for the length of the file. When we remove number of functions from the input data of the initial experiment static_outgoingCallRels_6 is at the root of the resulting tree. This suggests that number of functions is somehow related to outgoing calls. However, such a conclusion is premature considering the displayed complex dependencies between the various metrics.

In experiment four and five we showed, that it is possible to predict future defect-density with data mining techniques to an extend that is useful for an engineer or the management. We therefore can accept Hyp 2.

However, as we have seen in the other experiments, the relationships between the various metrics are complex. The sheer amount of data makes it impossible to intuitively understand the underlying decisions of a classifier by just looking at a generated tree. This leeds to the partly rejection of Hyp 3.

5. CONCLUSIONS AND FUTURE WORK

Our long term goal is to provide software project teams with tools allowing a manager to invest resources proactively (rather than reactively) to improve software quality before delivery. A key factor of such tools is the capability to predict defect densities in software modules such as source files or classes.

In this paper we specifically investigated the application of data mining on a number of source code, modification, and defect measures to test their applicability for defect prediction. The focus of our work is more on the understanding of the factors that lead to defects than the actual defect prediction. For this we stated a set of hypotheses that we addressed in a series of experiments with data from seven releases of the content and layout modules of the Mozilla open source project.

The data mining experiments showed, that a decision tree learner (J48) can produce reasonable results with various sets of input data. Regarding our hypotheses:

- We were able to *predict defect densities with acceptable accuracies* with metrics from the same release and therefore accepted Hyp 1.
- We found that *lines of code has little predictive power* with regard to defect-density therefore rejected Hyp 1a and Hyp 1b. In general, *size metrics such as number of functions are of little value for predicting defect densities*.
- We were able to *predict defect-density with satisfactory accuracy* by using evolution data (*e.g.,* number of modification reports) therefore accepting Hyp 2.

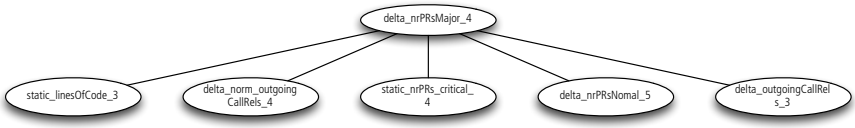

Figure 7: Top levels of decision tree resulting from Exp 5

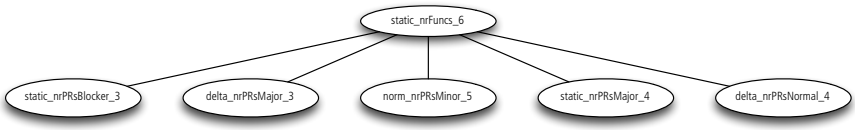

Figure 8: Top levels of decision tree resulting from Exp 6

- Due to complex relationships between the various metrics we could *only partly identify factors that lead to high defect-density*. This resulted in the partly rejection of Hyp 3.
- We found that *change couplings are of little value for the prediction of defect-density* therefore we rejected Hyp 4a and Hyp 4b.

Future work is concerned with including detailed measures of modifications (*e.g.,* number of statements changed) and defects (*e.g.,* bug status information) in our experiments. In addition, we also plan to take into account the various source code complexity measures, such as McCabe's cyclomatic complexity or the Halstead complexity measures. With this additional information we can gain deeper insights into the internals of the implementation as well as the past defects and modifications that caused increase as decreases of defect densities in source files and classes.

Another area of future work is to use other data mining techniques and conduct our experiments with additional case studies from the open source community as well as industrial software systems.

6. ACKNOWLEDGMENTS

We thank Harald Gall, Peter Vorburger, Beat Fluri and the anonymous reviewers for their valuable input. This work was supported by the Swiss National Science Foundation.

7. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

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

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[4] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[6] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[7] 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.

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

[9] 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 The Seventh International Symposium on Software Reliability Engineering*, page 364, Washington, DC, USA, 1996. IEEE Computer Society.

[10] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, 1992.

[12] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM Press.

[13] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005. IEEE Computer Society Press.

[14] I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann Publishers, 1999.

[15] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.