Predicting Defects and Changes with Import Relations

Adrian Schröter
Saarland University
Saarbrücken, Germany
schroeter@st.cs.uni-sb.de

Abstract

Lowering the number of defects and estimating the development time of a software project are two important goals of software engineering. To predict the number of defects and changes we train models with import relations. This enables us to decrease the number of defects by more efficient testing and to assess the effort needed in respect to the number of changes.

1. Introduction

”Why do people produce error-prone code?” this question motivated many studies like [1, 2, 3] and is one of the central questions of software engineering. To answer it, we must search for properties of a program or its development process that commonly correlate with defect density; in other words, once we can find such properties, we can explore their effects.

Another area of interest we want to contribute to is effort estimation. For a manager it is of utmost importance to know how long a software project will take. A way to asess the effort is the number of changes a software project is to undergo. Simply put, once we can predict the number of changes, we have a better understanding of the needed effort.

In this work, we take advantage of the study conducted by Schröter et al. [3] and learn from history which import relations correlated with defects and changes. In other words: Does using the library A increase the number of defects or changes? With such predictions, we can decrease the risk of defects and asess the effort spend on the software project.

In remainder of this work, we first give the idea behind our predictions (Section 2). Next, we describe how to collect the necessary data (Section 3). From this data, we train models and make our predictions (Section 4). At last we close with some future work (Section 5).

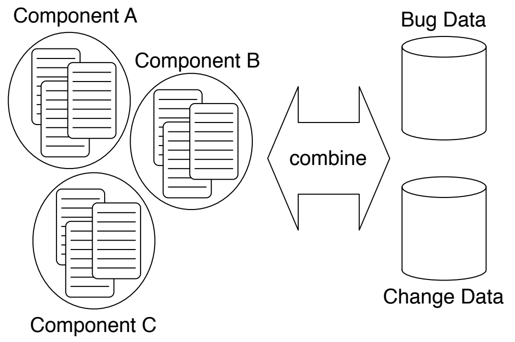

Figure 1. Combine files with bug/change data and assign them to components.

2. The Idea

The idea is to find properties, which influences the number of defects and/or the number of changes. We come to believe that one of those properties is the problem domain. But how can we measure or categorize the problem domain? In programming languages like JAVA we can take advantage of the import structure. This import structure reflects usage relation of single components and therefore gives us an insight into the problem domain.

defects. Why does the problem domain effects the number of defects? As we know defects are not spread equally over a programm. Take for instance a `ui` and `compiler` component as described in [3]. We see that on the ECLIPSE data set, components related to the `compiler` are correltated with defect and `ui` with success.

Changes. The influence of the problem domain on the number of changes is similar to the number of defects. On the one hand each defect needs to be fixed via a change thus having a direct influence on the number of changes. On the other hand `compiler` components use to be more complex than `ui` components. Such

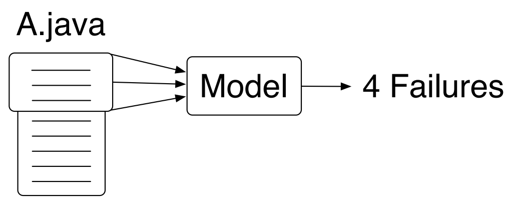

Figure 2. Use imports of a JAVA-file as input to predict number of defects.

components usually take more time to maintain, which simply results in more changes.

3. Get the Data

Before we can conduct our predictions we need to gather the necessary data. As Figure 1 illustrates we combine JAVA-files with their bug and change data. Thus we obtain the defect and change count for each file. Afterwards we group each file to components, like `compiler` and `ui`. In the following we explain where we got the data.

Files. Our training data consists of ECLIPSE version 3.0 source code, whereas we make the prediction on the newest source available to the project.

defect-data. To combine the training source code with defect counts, we use the ECLIPSE defect data provided by www.st.cs.uni-sb.de [2].

Change-data. We extract the total number of changes made to each file contained in the ECLIPSE 3.0 version archive.

Features. In order to obtain the feature set we extract the import statement of each JAVA-file in ECLIPSE version 3.0.

Knowing the data generation process we can describe how we build the prediction model and make the predictions.

4. Make the Prediction

Since the bug and change data we gathered cover more than two months we need to scale the dependent variable (bug or change count respectively). As the study of Schröter et al. [3] showed support vector machines perform best with import statements as feature set. Moreover, they are able to obtain accurate results across different versions. Hence, we can train the support vector machine with the data of one ECLIPSE version and apply the model on a later version without losing accuracy. Therefore, we opt to train a support vector machines as prediction models.

To determine the best configuration for the support vector machine we perform a ten fold cross validation. We asess the validation results with Spearman rank correlation and Pearson correlation coefficients.

The prediction is straight forward as Figure 2 shows. We extract as mentioned previously the imports statements of JAVA-files. Subsequently, we remove all import statements, which do not occur in the ECLIPSE version 3.0 data and predict the number of defects or changes respectively.

At last we simply sum up the number of changes and defects componentwise to obtain our final result.

5. Future Work

Since our predictions will not be perfect, we continue to look for more properties correlating with either defects or changes. Features we would like to look into include inheritance relations (does sub-classing from a class C increase defect-proneness?), part-of relations (does including C as a part influence the likelihood of defect?), or general design metrics such as depth of inheritance or number of subclasses.

References

[1] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. November 2005.

[2] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... (short paper). In *Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, pages 18–20, September 2006.

[3] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–27, September 2006.