Expert Systems with Applications 36 (2009) 9986–9990

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Data mining source code for locating software bugs: A case study in telecommunication industry

Burak Turhan *, Gozde Kocak, Ayse Bener

Dept. of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey

ARTICLE INFO

Keywords:
Software testing
Defect prediction
Software bugs
Case study

ABSTRACT

In a large software system knowing which files are most likely to be fault-prone is valuable information for project managers. They can use such information in prioritizing software testing and allocating resources accordingly. However, our experience shows that it is difficult to collect and analyze fine-grained test defects in a large and complex software system. On the other hand, previous research has shown that companies can safely use cross-company data with nearest neighbor sampling to predict their defects in case they are unable to collect local data. In this study we analyzed 25 projects of a large telecommunication system. To predict defect proneness of modules we trained models on publicly available Nasa MDP data. In our experiments we used static call graph based ranking (CGBR) as well as nearest neighbor sampling for constructing method level defect predictors. Our results suggest that, for the analyzed projects, at least 70% of the defects can be detected by inspecting only (i) 6% of the code using a Naïve Bayes model, (ii) 3% of the code using CGBR framework.

1. Introduction

Software testing is one of the most critical and costly phases in software development. Project managers need to know "when to stop testing?" and "which parts of the code to test?". The answers to these questions would directly affect defect rates and product quality as well as resource allocation (i.e. experience of test staff, how many people to allocate for testing) and the cost.

As the size and complexity of software increases, manual inspection of software becomes a harder task. In this context, defect predictors have been effective secondary tools to help test teams locate potential defects accurately (Menzies, Greenwald, & Frank, 2007). These tools are built using historical defect databases and are expected to generalize the statistical patterns for unseen projects. Thus, collecting defect data from past projects is the key challenge for constructing such predictors.

In this paper, we share our experience for building defect predictors in a large telecommunication system and present our initial results. We have been working with the largest GSM operator (~70% market share) in Turkey, Turkcell, to improve code quality and to predict defects before the testing phase. Turkcell is a global company whose stocks are traded in NYSE and operates in Turkey, Azerbaijan, Kazakhstan, Georgia, Northern Cyprus and Ukraine with a customer base of 53.4 million. The underlying system is a standard 3-tier architecture, with presentation, application and data layers. Our analysis focuses on the presentation and application layers. However, the content in these layers cannot be separated as distinct projects. We were able to identify 25 critical components, which we will refer to as project throughout this paper.

We used a defect prediction model that is based on static code attributes like lines of code, Halstead and McCabe attributes. Some researchers have argued against the use of static code attributes claiming that their information content is very limited (Fenton & Neil, 1999). However, static code attributes are easy to collect, interpret and many recent research have successfully used them to build defect predictors (Menzies, Greenwald et al., 2007; Menzies, Turhan, Bener, & Distefano, 2007; Turhan & Bener 2007; Turhan & Bener 2008). Furthermore, the information content of these attributes can be increased i.e. using call graphs (Kocak, Turhan, & Bener, 2008a; Kocak, Turhan, & Bener, 2008b). Kocal et al. show that integrating call graph information in defect predictors decreases their false positive rates while preserving their detection rates. Previously, Turkcell did not use company-wide policies for collecting and analyzing such metrics. In our research, we have collected these metrics from the abovementioned 25 projects. We have also collected the static call graphs for these projects.

The collection of static code metrics and call graphs can be easily carried out using automated tools (Menzies, Greenwald et al., 2007; Menzies, Turhan et al., 2007; Turhan & Bener 2008). However, as we mentioned earlier, matching these measurements to software components is the most critical factor for building defect predictors. Unfortunately, in our case, it was not possible to match

* Corresponding author. Tel.: +90 212 3597227; fax: +90 212 2872461.
E-mail address: turhanb@boun.edu.tr (B. Turhan).

0957-4174/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2008.12.028

past defects with the software components in the desired granularity, module level, where we mean the smallest unit of functionality (i.e. java methods, c functions). Previous research in such large systems use either component or file level code churn metrics to predict defects (Bell, Ostrand, & Weyuker 2006; Nagappan & Ball, 2006; Ostrand & Weyuker, 2002; Ostrand, Weyuker, & Bell 2005; Ostrand, Weyuker, & Bell, 2004; Ostrand, Weyuker, & Bell, 2007; Zimmermann & Nagappan, 2006). The reason is that file level is the smallest granularity level that can be achieved. For example, Nagappan, Ball and Zimmermann analyze Microsoft software in component level and Ostrand, Weyuker and Bell analyze AT&T software in file level to report effective predictors used in practice. However, defect predictors become more precise as the measurements are gathered from smaller units (Ostrand et al., 2007).

Therefore, we decided to use module level cross-company data to predict defects for Turkcell projects (Menzies, Turhan et al., 2007). Specifically, we have used module level defect information from Nasa MDP projects to train defect predictors and then obtained predictions for Turkcell projects. Previous research have shown that cross-company data gives stable results and using nearest neighbor sampling techniques further improves the prediction performance when cross-company data is used (Menzies, Greenwald et al., 2007; Menzies, Turhan et al., 2007; Turhan & Bener, 2008). Our experiment results with cross-company data on Turkcell projects, estimate that we can detect 70% of the defects with a 6% LOC investigation effort.

While nearest neighbor algorithm improves the detection rate of predictors built on cross-company data, false alarm rates remain high. In order to decrease false alarm rates, we included the call graph based ranking (CGBR) framework in our analysis based on our previous research. We used graph based ranking (CGBR) framework (Kocak et al., 2008a; Kocak et al., 2008b) to software modules. Using CGBR framework improved our estimated results such that the LOC investigation effort decreased from 6% to 3%.

The rest of the paper is organized as follows: In section 2 we briefly review the related literature, in Section 3 we explain the project data. Section 4 explains our rule-based analysis. Learning based model analysis is discussed in Section 5. The last section gives conclusion and future direction.

2. Related work

Ostrand and Weyuker have been performing similar research for AT&T and they also report that it is hard to conduct an empirical study in large systems due to difficulty in finding the relevant personnel and the high cost of collecting and analyzing data (Ostrand & Weyuker, 2002). Nevertheless, there are notable research in large software systems (Adams, 1984; Basili & Perricone, 1984; Bell et al., 2006; Fenton & Ohlsson, 2000; Menzies, Greenwald et al., 2007; Menzies, Turhan et al., 2007; Ostrand & Weyuker, 2002; Ostrand et al., 2004; Ostrand et al., 2005; Ostrand et al., 2007). Fenton and Ohlsson presented results of an empirical study on two versions of a large-scale industrial software, which showed that the distribution of the faults and failures in a software system can be modeled by Pareto principle (Fenton & Ohlsson, 2000). They claimed that neither size nor complexity explain the number of faults in a software system. Other researchers found interesting results showing that small modules are more fault-prone than larger ones (Koru & Liu, 2005a; Koru & Liu, 2005b; Malaiya & Denton, 2000; Zhang 2008). Our results will also show evidence in favor of this fact.

As mentioned, Ostrand, Weyuker and Bell also worked with large telecommunications software systems in AT&T (Bell et al., 2006; Ostrand & Weyuker, 2002; Ostrand et al., 2004; Ostrand et al., 2005; Ostrand et al., 2007). They predicted fault-prone files of the large software system by using a negative binominal regression model. They report that their model can detect 20% of the files that contain 80% of all faults. Similarly, Nagappan, Ball and Zimmermann analyzed several Microsoft software components using static code and code churn metrics to predict post-release defects. They observed that different systems could be best characterized by different sets of metrics (Nagappan & Ball, 2006; Zimmermann & Nagappan, 2006).

Our work differs at a large extent from previous work. Ostrand, Weyuker and Bell carried out the most similar work to this research, where they used file level measurements as a basic component. However, we prefer using modules, since modules provide finer granularity. They have collected data from various releases of projects and predict post-release defects, whereas we have data from single release of 25 projects and we try to predict pre-release defects.

project	# modules	features	LOC
Trcll 1	572	29	6206
Trcll 2	3089	29	80941
Trcll 3	3963	29	45323
Trcll 4	260	29	5803
Trcll 5	2698	29	53690
Trcll 6	155	29	4526
Trcll 7	120	29	5423
Trcll 8	4320	29	79114
Trcll 9	350	29	10221
Trcll 10	2076	29	61602
Trcll 11	57	29	2485
Trcll 12	497	29	9767
Trcll 13	189	29	5425
Trcll 14	132	29	2965
Trcll 15	1826	29	36280
Trcll 16	1106	29	42431
Trcll 17	460	29	6933
Trcll 18	345	29	10601
Trcll 19	273	29	6258
Trcll 20	339	29	3507
Trcll 21	150	29	1971
Trcll 22	19921	29	215265
Trcll 23	4518	29	51273
Trcll 24	232	29	10135
Trcll 25	347	29	4880

Fig. 1. Turkcell datasets used in this study.