The Journal of Systems and Software 82 (2009) 1793–1803

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Increasing diversity: Natural language measures for software fault prediction

David Binkley [a], Henry Feild [b], Dawn Lawrie [a,*], Maurizio Pighin [c]

[a] Loyola College Baltimore, MD 21210, USA
[b] University of Massachusetts, Amherst, MA 01003, USA
[c] Universitá degli Studi di Udine, Italy

ARTICLE INFO

Article history:
Available online 26 June 2009

Keywords:
Information retrieval
Code comprehension
Fault prediction
Linear regression models
Empirical software engineering

ABSTRACT

While challenging, the ability to predict faulty modules of a program is valuable to a software project because it can reduce the cost of software development, as well as software maintenance and evolution. Three language-processing based measures are introduced and applied to the problem of fault prediction. The first measure is based on the usage of natural language in a program's identifiers. The second measure concerns the conciseness and consistency of identifiers. The third measure, referred to as the QALP score, makes use of techniques from information retrieval to judge software quality. The QALP score has been shown to correlate with human judgments of software quality.

Two case studies consider the language processing measures applicability to fault prediction using two programs (one open source, one proprietary). Linear mixed-effects regression models are used to identify relationships between defects and the measures. Results, while complex, show that language processing measures improve fault prediction, especially when used in combination. Overall, the models explain one-third and two-thirds of the faults in the two case studies. Consistent with other uses of language processing, the value of the three measures increases with the size of the program module considered.

1. Introduction

This paper studies the application of natural language processing techniques to the problem of fault prediction. Detecting fault prone code early, regardless of software life-cycle phase, allows for the code to be fixed at lower cost; thus, a good fault predictor helps to lower development and maintenance costs. For example, cost savings may come from focusing testing-effort on certain parts of the software, restructuring code, or augmenting documentation. Further motivation comes from Koru and Tian who observe that "software products are getting increasingly large and complex, which makes it infeasible to apply sufficient reviews, inspections, and testing on all product parts given finite resources" (Koru and Tian, 2007), highlighting the need for good fault prediction.

A number of studies have found correlations between structural characteristics of software modules and problems, such as change or defect proneness (Bell et al., 2006; Fenton and Ohlsson, 2000; Gyimóthy et al., 2005; Kokol et al., 2001; Koru and Tian, 2007; Menzies et al., 2007; Munson and Khoshgoftaar, 1992). Example structural measures include lines of code, operator counts, nesting depth, message passing, coupling, information flow-based cohesion, depth of inheritance tree, number of parents, number of previous releases in which the module occurred, and number of faults detected in the module during the previous release (Bell et al., 2006; Ferenc et al., 2002). However, it has been observed that there is need for more sophisticated measures. For example, Nortel Networks and IBM engineers observe that the most troublesome modules are not the ones with the highest structural-measure values (Koru and Tian, 2007); thus, observing the need for more sophisticated techniques.

In addition to greater sophistication, in recent work with structural code measures, Menzies et al. argued that the particular set of measures used in fault prediction is less important than having a sufficient pool to choose from (Menzies et al., 2007). Diversity in this pool is important. For example, many existing measures are strongly correlated with lines of code. One avenue to improve fault predictors is the search for additional measures not correlated with those in the existing pool.

Until recently, the semantic information contained in the natural language of a program (in particular, its identifiers) has gone underutilized in software engineering (perhaps owing to the origin of many analyses in the compiler construction field). The measures considered herein augment those that use structural characteristics by incorporating the semantics of natural language. This complements the structural information used in most measures by providing an orthogonal view of the source code. One view of these

* Corresponding author.
E-mail addresses: binkley@cs.loyola.edu (D. Binkley), hfeild@cs.umass.edu (H. Feild), lawrie@cs.loyola.edu (D. Lawrie), maurizio.pighin@uniud.it (M. Pighin).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.06.036

measures is an attempt to capture the information and intuitive notion of *well written* code. While instructors have long advocated the use of *good* identifier names, it is language processing techniques that can quantify this value. To this end, two case studies show that the language-oriented techniques deserve future study in the challenging domain of software fault prediction.

Three natural language-oriented measures are studied in this paper. The first is the simplest: it measures the percentage of natural language used in a program's identifiers. The intuition is that identifiers composed of natural language words may be easier for an engineer to understand; thus, leading to fewer faults. The second measure is the percent of identifiers violating a variant of Deißenböck and Pizka's rules for *concise and consistent* identifiers (Deißenböck and Pizka, 2005). Ambiguity in the concepts associated with identifiers makes code harder to manipulate without introducing faults (in particular, those associated with concept misunderstanding). The third, referred to as the *QALP score*, is named after a project aimed at providing Quality Assessment using Language Processing (Lawrie et al., 2006). The QALP score measures correlations between the natural language use in a program's source code and its documentation. It thus attempts to quantify the notion of *well-documented code*.

To investigate the value of the three natural language-oriented measures in fault prediction, two case studies are considered – one using the open source program Mozilla and the other a proprietary program written for a business application in a mid-size enterprise. These studies assess the utility of the language-oriented measures in predicting fault-prone modules of source code. Of particular interest is the notion of diverse measures and their importance in fault prediction. Therefore, two models for each case study are presented. In one model, a single language-oriented measure is used in conjunction with other structural measures. In this case, the QALP score is used because it is the most complex of the three. The second model incorporates all three natural language-oriented measures.

The primary contributions of this paper are the following:

1. **Natural language measures.** First and foremost, the paper proposes the use of measures completely unrelated to structural aspects of source code as a means of improving diversity in the pool of fault prediction measures.
2. **Case studies.** It also considers two case studies that explore the usefulness of the proposed measures in fault prediction.

The remainder of the paper includes background information in Section 2. Then a description of the experimental setup of the two case studies is presented in Section 3. The two case studies are presented in Section 4, followed by a discussion of related work, implications of this research, and a summary in Sections 5–7.

2. Background

This section describes each of the proposed measures, providing motivation for their inclusion and describing their computation. All three measures use a common preprocessing step, word extraction, which is described first. Following the description of the measures is a brief overview of the two case-study subjects. Finally, a description of the statistical technique used is given.

2.1. Word extraction

Word extraction has two phases. The first extracts the identifiers from the source and the second splits them into their constituent words (Feild et al., 2006; Lawrie et al., 2006). The first phase is implemented as a simple lex-based scanner (essentially implementing a simple Island Grammar (Moonen, 2001)).

The goal of the second phase is to split the extracted identifiers into 'words'. Each word is a sequence of characters to which some meaning may be associated. The need for splitting comes from identifiers that are made up of multiple "*words*" fused together (*e.g.*, rootcause). Words are often demarcated by word markers (*e.g.*, using CamelCaseing or under_scores). For example, the identifiers spongeBob and sponge_bob both contain the demarcated words sponge and bob. Such words are referred to as *hard words*.

When words are not explicitly demarcated, a splitting algorithm is used to divide each *hard word* into its constituent words. One such algorithm is a greedy algorithm that recursively searches for the longest dictionary prefix and suffix of (the remaining part of) an identifier (Feild et al., 2006). For example, consider the code

```
/* Sponge Bob needs to be given a bath */
bath(spongebob);
```

The greedy algorithm retains the hard word bath, but decomposes the hard word spongebob into sponge and bob. Words that are identified by the splitting algorithm are referred to as *soft words* (*i.e.*, bath, sponge, and bob). Thus, a hard word is made up of one or more soft words. Soft words form the atomic entities used by the three measures.

2.2. Use of natural language

The first measure, *percent natural language*, is the number of unique soft words found in the dictionary divided by the total number of unique soft words found in the code. It is hypothesized that the more natural language words used in identifiers, the easier the code will be to understand. This in turn is expected to lead to fewer faults. To determine whether or not each soft word comes from a natural language, it is looked up in a dictionary. When multiple natural languages are found in the source code, multiple dictionaries are used. Finally, the precision of this measure is somewhat dependent on the accuracy of the splitting algorithm. The accuracy of the splitter used in the experiments is about 76% (Feild et al., 2006) when compared to a human oracle. In terms of the percent natural languages found in a program's identifiers, the imperfectness of the splitting algorithm is expected to cause slightly, but uniformly, inflate scores; thus, this inflation is not expected to have a significant effect on the studies' results.

2.3. Conciseness and consistency

The second measure is based on the percentage of identifiers that violate syntactic conciseness and consistency rules (Lawrie et al., 2006). Such violations potentially lead to confusion as to the concepts represented by identifiers and thus may make the code more fault prone. The rules are based on Deißenböck and Pizka's formal model for well-formed identifier naming (Deißenböck and Pizka, 2005).

Deißenböck and Pizka's rules include three requirements: two related to identifier consistency (involving homonyms and synonyms) and one related to identifier conciseness. These three are formalized as follows: an identifier *i* is a homonym if it represents more than one concept from the program (*e.g.*, the identifier file in Fig. 1a). Two identifiers *i1* and *i2* are synonyms if the concepts associated with *i1* have a non-empty overlap with the concepts associated with *i2* (*e.g.*, the identifiers file and file_name share the concept *file name* in Fig. 1b). The presence of homonyms or synonyms indicate inconsistent naming of concepts in a program and thus violate Deißenböck and Pizka's identifier consistency rule. Finally, an identifier *i* for concept *c* is concise if no concept less general than *c* is represented by another identifier. For example, the identifier position most directly corresponds to the concept *posi-*