Software Fault Prediction using Language Processing

David Binkley[†] **Henry Feild**[†] **Dawn Lawrie**[†] **Maurizio Pighin**[‡]
[†]Loyola College
Baltimore MD
21210-2669, USA
{binkley, hfeild, lawrie}@cs.loyola.edu

[‡]Universita' degli Studi di Udine
Udine
33100 Italy
maurizio.pighin@uniud.it

Abstract

Accurate prediction of faulty modules reduces the cost of software development and evolution. Two case studies with a language-processing based fault prediction measure are presented. The measure, refereed to as a QALP score, makes use of techniques from information retrieval to judge software quality. The QALP score has been shown to correlate with human judgements of software quality. The two case studies consider the measure's application to fault prediction using two programs (one open source, one proprietary). Linear mixed-effects regression models are used to identify relationships between defects and QALP score. Results, while complex, show that little correlation exists in the first case study, while statistically significant correlations exists in the second. In this second study the QALP score is helpful in predicting faults in modules (files) with its usefulness growing as module size increases.

Keywords: information retrieval, code comprehension, fault prediction, empirical software engineering

1 Introduction

This paper studies the application of information retrieval techniques to the problem of fault prediction. Detecting fault prone code early, regardless of software lifecycle phase, allows for the code to be fixed at lower cost; thus, a good fault predictor helps to lower development and maintenance costs. 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"[10], 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 [2, 7, 9, 10, 16, 19]. 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 [10]; thus, there is a need for more sophisticated techniques.

This paper considers the application of such a technique: it applies an Information Retrieval (IR) based technique to the problem of fault prediction. One motivation for this is IR's focus on natural language (*e.g.*, the words used to make up the identifiers). Incorporating the semantics of natural language complements the structural information in most metrics presently used in fault prediction. Two case studies show that IR-based techniques deserve future study in the challenging domain of software fault prediction.

Historically, IR has been applied to unstructured text (as opposed to the structured information used by database management systems). Recently, IR techniques, which can select relevant documents from large collections, "have proven useful in many disparate areas, including the management of huge scientific and legal literature, office automation, and to support complex software engineering projects"[1].

Although IR tools have focused on the analysis of prose; many of the techniques developed are applicable to arbitrary text documents, including source code. For example, Antoniol et al. and Marcus et al. independently analyzed comments and variables to (re)establish links between source code and its documentation [1, 15]. When considering source code, potential "documents" include source files, classes, or functions.

Many fault predictors focus on the maintenance phase of the software life cycle (*e.g.*, the work of Ostrand and Weyuker [2, 20]). This allows the fault predictor to make use of information about past faults in predicting current faults. More general fault prediction makes use of a plethora of structural measures in order to predict faults, even in the absence of a fault history (*e.g.*, the work of Menzies et al. [16]). 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 the module occurred in, and number of faults detected in the module during the previous release [2, 8].

In a recent paper, Menzies et al. argue that the particular set of structural measures used by many fault predictors is less important than having a sufficient pool to choose from [16]. 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 that are not correlated with those in the existing pool. The measure considered herein is one example.

The IR based measure studied in this paper, referred to as a *QALP score*, is named after a project aimed at providing Quality Assessment in the large using Language Processing [14]. The QALP score was developed as part of a tool aimed at leveraging human insight, intuition, and judgment to assess code quality in the large. It does so by extracting aspects of a system for consideration. From these, an engineer can gain an understanding of the system's overall quality. This assessment is useful, for example, in an out-sourcing environment to evaluate the expected cost of maintaining delivered code.

Although the QALP score was not developed for use in fault prediction, its focus on quality makes it potentially well suited to the task. In particular, it focuses on natural language, and thus indirectly the comprehensibility of the code. To investigate the value of the QALP score in fault prediction, two case studies are presented–one using the open source program Mozilla and the other a proprietary program. These studies assess the utility of the QALP score in predicting fault-prone modules of source code.

The remainder of the paper includes background information in Section 2, followed by a description of the experimental setup of the two case studies in Section 3. The primary contribution of the paper is the two case studies themselves, presented in Section 4, which are followed by a discussion of related work and a summary in Sections 5 and 6.

2 Background

This section first provides background information on the computation of the QALP score and the information retrieval concepts that underly it. Next, it presents a brief overview of the two software packages examined in the case studies. Finally, a description of the statistical techniques used is given.

2.1 QALP Score

The QALP score describes the similarity between a module's comments and its code. It is computed using *cosine similarity*, a technique developed for use in IR tools to retrieve *documents* relevant to a query [24]. In IR the term *document* refers to any cohesive unit of text and is usually the artifact returned as the result of a query. One popular method for finding relevant documents uses a *vector space model*. Such a model considers each word as a separate *dimension* in an n-dimensional vector space. The *similarity* between two documents is then defined as the cosine of the angle between their two vectors. Given that a query can also be expressed as a vector, documents can be ranked according to their cosine similarity to the query. The closer the cosine of the angle is to 1.0, the closer the match.

Several techniques are applied to improve cosine similarity. One employs a stop-list: a collection of words not thought to be relevant to any query. For example, in English, words such as '*the*' and '*an*' are stop-list words. In the context of this project, the comments are (assumed to be) written in natural language, so they are stopped using an appropriate natural language stop-list (for code with English comments, a standard English-language stop-list is used).

A second technique, *stemming*, reduces the dimensionality of the vector space by eliminating word suffixes; thus, ignoring the particular form of a word. For example, the 'stem' of '*run*', '*running*', and '*runs*' is '*run*'. Since IR uses exact matches of words, stemming improves document matching.

The final technique weights the words in the vector space. The QALP tool uses the standard method *term frequency−inverse document frequency* (*tf-idf*) [22], which provides a method for weighting the importance of a given word (called a *term*) to a document relative to the frequency of the term in the entire collection. The weighting takes into account two factors: term frequency in the given document and inverse document frequency of the term in the whole collection. In short, term frequency in a document shows how important the term is in that document. Document frequency of the term shows how common the term is. A high weight using *tf-idf* is achieved by a term that occurs much more than the average in a document, but is rare in the entire collection.

To compute a QALP score, two separate documents are constructed from each module. One contains the comments of the module and the other the code of the module. The comments are stopped and stemmed. The code is only stopped using a special stop-list that includes frequently used words not indicative of the concepts present in the code [14]. For example, the stop-list for C includes keywords (*e.g.*, `while`), predefined identifiers (e.g., `NULL`), library function and variable names (e.g., `strcpy` and

errno), and all identifiers that consist of a single character. The resulting score is a real number between zero, no similarity, and one, 'perfect' similarity.

After stopping, program identifiers are split. The need for splitting comes from identifiers made up from multiple words fused together (*e.g.*, rootcause). To facilitate matching code with the comments, such identifiers must be divided into their constituent parts [6, 13]. Herein, these parts are referred to as "*words*" – sequences of characters with which some meaning may be associated. Words are often demarcated by the use of word markers (*e.g.*, CamelCaseing or under_scores). For example, the identifiers spongeBob and sponge_bob both contain the demarcated words sponge and bob.

When words are not explicitly demarcated, a splitting algorithm is used to divide each identifier into its constituent words. A greedy algorithm that recursively searches for the longest dictionary prefix and suffix of (the remaining part of) an identifier is one option [6]. For example, consider the code

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

The greedy algorithm decomposes givebath into give and bath and spongebob into sponge and bob.

In addition to splitting the code, the comments are stemmed. In the example above, stemming replaces given with its present tense form give. After stemming and splitting, the similarity between the comments and the code comes through in the QALP score as measured using the cosine similarity technique. Note that without stemming the comments and splitting the identifiers, there are no 'words' in common between the two; thus, the QALP score would be zero indicating a lack of overlap.

In the study QALP scores are computed by the search engine Yari, developed by Victor Lavrenko at the Center for Intelligent Information Retrieval [12]. One advantage of using Yari is that the search engine is freely available for research purposes along with its source code, so that new applications can be developed based on a collection indexed by Yari. In addition to the ability to build retrieval collections from arbitrary text, additional functionality was been included in Yari. For example, there are several similarity functions including cosine similarity and part-of-speech recognition functions that enable the recognition of nouns in natural English.

2.2 Study Subjects

Two programs were used in the case studies: version 1.6 of the open source browser, Mozilla, and a proprietary program, referred to as $\mathcal{MP}$. These programs were chosen based on their size and the availability of defect (fault) data. In the case of Mozilla this data was extracted by examining the bug database maintained by Bugzilla which assigns each bug to a set of classes [9]. For $\mathcal{MP}$ fault data was collected in a similar in-house database in which two kinds of bugs are recorded: those revealed during the integration test phase and those generated by faults "in the field" during operation. For both, the database stores the module containing the fault, its date, a description of fault, and the customer (if present).

In more detail, Mozilla is coded primarily in C++, with parts in C and Java. It contains 3,004,824 *LoC* (lines of code as measured by the UNIX utility wc) and 2,383,034 *SLoC* (source lines of code as measured by SLOCCount [26]). *SLoC* includes only non-comment, non-blank lines of code. By language, Mozilla contains 1,549,636 *SLoC* of C++, 829,814 *SLoC* of C, and 3,584 *SLoC* of Java. The second case study, $\mathcal{MP}$, is written exclusively in C and has 454,609 *LoC* and 282,246 *SLoC*. The software was written for a business application in a mid-size enterprise. It runs under UNIX and interfaces with Infromix database using a specific access library (All-II).

As described by Ferenc et al., the defect data for Mozilla contains the number of defects found in the 3,677 C++ classes [9]. The data was collected by examining the bugs from bugzilla. Each bug report included a list of lines from one or more files that had been modified to fix the bug. By examining the source code, each bug was assigned to one or more classes; thus, the final outcome was a bug count per class. If the bug fix changed more than one class then each effected class had its count incremented. Classes generated on-the-fly were ignored. In all, more than half of the classes (1,850 of them) contained no bugs, and about one fifth (666 of them) contained only one bug.

For $\mathcal{MP}$ defect data was available for each of the 1,161 C source files. The defects were tracked in a database. All defects had an associated date and module. Given that the program was developed by a team of 20 engineers, one programmer from the team was assigned to fix a particular defect. For this reason, many programmers worked on the same module over the 10 to 12 year life-cycle of the project.

2.3 Statistical Tests

Linear mixed-effects regression models are used to analyze the data [25]. Such models easily accommodate unbalanced data, and, consequently, are ideal for this analysis. These statistical models allow the identification and examination of important explanatory variables associated with a given response variable.

The construction of a linear mixed-effects regression model starts with a collection of explanatory variables and a number of interaction terms. The interaction terms allow the effects of one explanatory variable to change de-

pending upon the value of another explanatory variable. Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Some non-significant variables and interactions are retained to preserve a hierarchical well-formulated model [18]. To provide a measure of model quality, a p-value and the coefficient of determination R^2 are reported with each model. The coefficient of determination can be interpreted as the percentage of the variation in the number of defects that is explained by the model.

In the mixed-effect models, computing a standard t-value for each comparison and then using the standard critical value increases the overall probability of a Type I error. Thus, Bonferroni's correction is made to the p-values to correct for multiple testing. In essence each p-value is multiplied by the number of comparisons and the adjusted p-value is compared to the standard significance level (0.05) to determine significance. Bonferroni's correction is chosen because it is a rather conservative test.

3 Experimental Setup

This section describes the two steps taken to setup the data collection for the case studies. In order to focus these studies on understanding the value and viability of using the QALP score in fault prediction, only three metrics are considered: the QALP score and the two structural measures *LoC* (the total lines of code) and *SLoC* (the total source lines of code, that is, non-comment, non-blank lines of code). The first step computes the structural measures for each module (each Mozilla class and each $\mathcal{MP}$ file).

The second step, computing the QALP score, includes three preprocessing phases, as pictured in Figure 1. The first phase (from Figure 1a to 1b) breaks the source into modules. Natural modules include functions, classes, and files, but the QALP score computation is not tied to any given definition. In this phase, the method of reporting the defect data determined the type of module used when computing the QALP score. Often, this step can be omitted as the source is physically laid out by module. $\mathcal{MP}$ source, for example, was organized by file, which matches the defect data. For Mozilla, some files contain multiple classes. In such cases, Phase 1 gathers together the code associated with a single class into a single file used to hold the source of the particular module.

The second phase separates each model into comments and code as illustrated in Figure 1c. This is done using *src2srcml* to insert XML tags throughout the code [4]. Once marked-up, the source is then run through a simple fact extractor to separate the comments and code. The comments include those that occur immediately before a class or function definition and those that appear within the class or function.

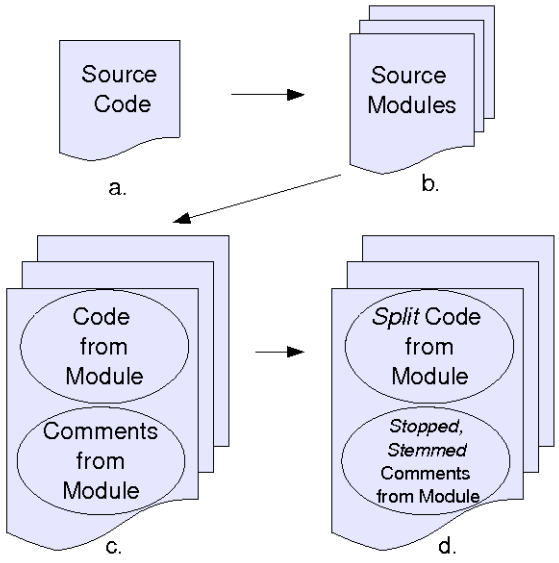

Figure 1. The preprocessing steps.

The final phase applies language processing techniques to improve the efficiency and the accuracy of the cosine similarity. In particular, as illustrated in Figure 1d, the identifiers are split and the comments are stopped and stemmed. Finally, Yari is used to compute the QALP score for each module.

4 Discussion of Results

The analysis begins with a simple graphical inspection followed by statistical models for each of the two case studies. It then concludes with a discussion comparing the two models. The graphical analysis, shown in Figure 2, plots the maximal QALP scores for all modules of a given defect count. For example, of all Mozilla classes with six reported defects, the maximal QALP score is just greater than 0.4.

An inverse relationship is expected between QALP score and defects where high quality modules (those with a high QALP score) are low in defects. As is evident in Figure 2, the expected pattern is seen where higher QALP scores form an *envelope* that appears as a 'line' sloping down and to the right. This envelope is more pronounced for Mozilla – the top line of Figure 2. The maximum QALP score of $\mathcal{MP}$, the lower line in Figure 2, also shows this relationship although it is less pronounced.

4.1 Mozilla

The statistical analysis first considers Mozilla. The initial linear mixed-effects regression model for predicting the defects in Mozilla begins with the explanatory variables

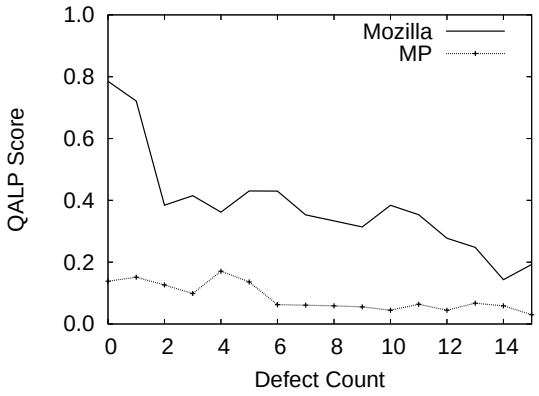

Figure 2. Maximum QALP score per defect count for both programs.

QALP score, denoted as *qs*, *LoC*, *SLoC*, and their interactions. From this model statistically insignificant terms (those having a p-value > 0.05) are removed. Dishearteningly, the QALP score is one of the terms removed. The last model (during the elimination) to include the QALP score is

$$\begin{aligned} \textit{defects} = 0.68 &- 0.15qs \\ &+ 7.3 \times 10^{-3}\, LoC \\ &- 6.8 \times 10^{-4}\, SLoC \\ &- 1.0 \times 10^{-6}\, LoC \times SLoC \end{aligned} \tag{1}$$

The negative coefficient for *qs* in this model would be good news except that the p-value for the QALP score of 0.829 is quite high; thus, QALP score is not significant in predicting the value of *defects*.

The final model, which includes only *LoC*, *SLoC*, and their interaction, is

$$\begin{aligned} \textit{defects} = 0.61 &+ 7.2 \times 10^{-3}\, LoC \\ &- 6.1 \times 10^{-4}\, SLoC \\ &- 1.0 \times 10^{-6}\, LoC \times SLoC \end{aligned} \tag{2}$$

Its coefficient of determination, $R^2 = 0.156$, is quite low (the model's p-value is < 0.0001). Thus, for Mozilla neither the QALP score nor the size measures prove to be good defect predictors.

An inspection of code for Mozilla generated three relevant insights as to why the QALP score proved an ineffective measure for predicting faults. First, many of the classes had few, if any reported defects and, second, there are very few comments. A complete absence of comments produces a QALP score of zero. When these zeros occur in classes with a wide range of faults, they produce statistical 'noise', which interferes with the ability of the statistical techniques to find correlations.

Second, many of the comments that did exist were either used to make up for a lack of *self documenting code* or were *outward looking*. In the first case, the code uses identifiers that are not easily understood out of context and comments are required to explain the code. In the second case, comments are intended for users of the code and thus ignore the internal functionality of the code. In both cases, the code and comments have few words in common, which leads to a low QALP score. For example, the code snippet in Figure 3 shows an example of both types of comments. This snippet determines whether there is anything other than whitespace contained in the variable mText. Unfortunately, it is not clear from the called function name, IsDataInBuffer, that it is simply a whitespace test. The first comment informs the reader of this; thus, the comment is compensating for a lack of self-documentation. The second comment is an outward looking comment, reflecting on the implications that the local code segment may have on the system as a whole.

The third insight follows from Mozilla being open source, which means that it includes many different coding practices and styles [10]. This is evident when inspecting samples of the code where identifier naming and general commenting are not done in a systematic fashion.

4.2 $\mathcal{MP}$

The statistical analysis of $\mathcal{MP}$ begins with the same set of explanatory variables as that of Mozilla. As seen in Figure 2, the maximum QALP scores for $\mathcal{MP}$ show a less pronounced downward-right trend than those of Mozilla. However, the final model (after removal of statistically insignificant terms) is substantially different. This model, given below in Equation 3, includes several interaction terms, meaning that no direct correlation between the explanatory variables and the response variable can be given.

$$\begin{aligned} \textit{defects} = -1.83 &+ qs(-2.4 + 0.53\, LoC - 0.92\, SLoC) \\ &+ 0.056\, LoC \\ &- 0.058\, SLoC \end{aligned} \tag{3}$$

The model's coefficient of determination ($R^2 = 0.614$, p-value < 0.0001) indicates that it explains just over 61% of the variance in the number of defects.

The model indicates that the QALP score plays a role in the prediction of faults. Of prime interest is the coefficient

```
// Don't bother if there's nothing but whitespace.
// XXX This could cause problems...
if (! IsDataInBuffer(mText, mTextLength))
break;
```

Figure 3. Mozilla**-1.6 Example Code Snippet**

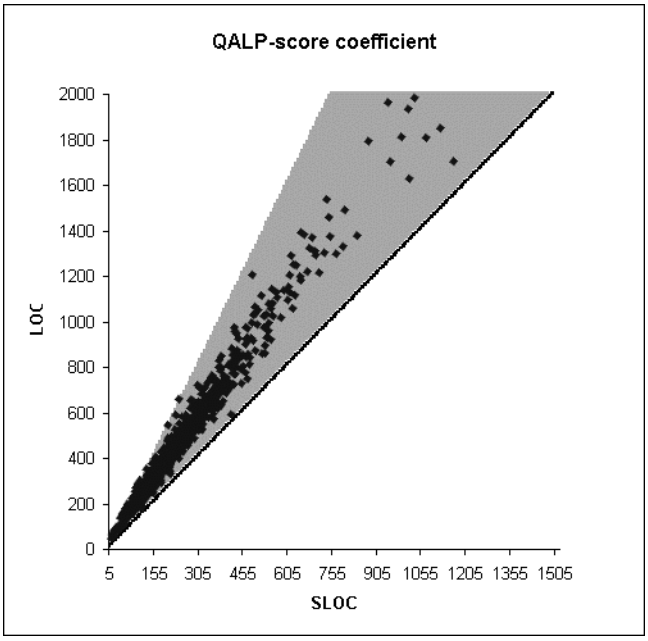

Figure 4. Break even point for coefficient of QALP score.

of the QALP score: $-2.4 + 0.53\,LoC - 0.92\,SLoC$. For higher QALP scores to be associated with lower defects, this coefficient has to be negative. In a simple model, free from interactions, this coefficient would be a constant and if less than zero the model would exhibit the desired correlation. In the presence of interactions, the value of the coefficient depends on other explanatory variables. Considering the coefficients in Equation 3, as *SLoC* increases, the coefficient of *qs* grows increasingly negative. A simplistic reading of this is that for larger modules, QALP score performs better. However, the coefficient for *LoC* in Equation 3, has the opposite interpretation: as *LoC* grows the coefficient grows increasingly positive.

Taken together these two make interpretating the model difficult. One approach is a graphical interpretation as shown in Figure 4 where *LoC* is graphed against *SLoC*. As *LoC* is bounded from below by *SLoC*, no point can occur below the solid 45 degree line. The gray area in the figure shows the region in which the coefficient of *qs* ($-2.4 + 0.53\,LoC - 0.92\,SLoC$) is dominated by $-0.92\,SLoC$. In this region, the coefficient of *qs* is negative (the region has a slight downward shift to account for the constant -2.4). Above the shaded region the term $+0.53\,LoC$ dominates and *qs* has a positive coefficient. The points in the graph represent the actual values for the modules of $\mathcal{MP}$. As all of these points fall in the shaded region, the coefficient of *qs* is negative over the range of combinations of *LoC* and *SLoC* actually observed.

A second method for interpreting models with multiple interactions uses a quantitative approach. The approach considers several values for one of the variables. In this case the ratio of *LoC* to *SLoC* is considered. Typical values to use include the mean together with the low and high end of the 95% confidence interval around the mean. For the model in Equation 3, these yield

$$\begin{aligned} LoC &= 1.665\,SLoC \quad \text{(lower bound)} \\ LoC &= 1.676\,SLoC \quad \text{(mean)} \\ LoC &= 1.687\,SLoC \quad \text{(upper bound)} \end{aligned}$$

Substituting these values in for *LoC* of Equation 3 yields the following three models

Using the Lower Bound

$$\begin{aligned} \textit{defects} = -1.83 + qs(&-2.4 - 0.047\,SLoC) \\ &+ 0.0345\,SLoC \end{aligned}$$

Mean

$$\begin{aligned} \textit{defects} = -1.83 + qs(&-2.4 - 0.041\,SLoC) \\ &+ 0.0351\,SLoC \end{aligned}$$

Using the Upper Bound

$$\begin{aligned} \textit{defects} = -1.83 + qs(&-2.4 - 0.035\,SLoC) \\ &+ 0.0357\,SLoC \end{aligned}$$

The positive coefficient of *SLoC* at the end of each equation supports the unsurprising result that an increase in module size (as measured in the non-comment, non-blank lines of code) brings an increase in defects. In this case at the rate of about three and a half per 100 *SLoC*. Due to the interaction with *qs*, this number should be taken as a rather rough estimate.

The coefficient of *qs* in all three equations is negative; meaning that QALP score has the desired slope. Furthermore, as module size increases, this coefficient grows increasingly negative; thus, QALP scores are better predictors for larger modules.

One final statistical model for $\mathcal{MP}$ is considered. In Equation 3 there is a complex interaction between *LoC* and *SLoC*. In particular the two have opposite signs. This makes general statements about the trend for larger or smaller modules difficult. In terms of the code, these two counts differ by two quantities: the number of comment lines and the number of blank lines. To better understand their impact, the number of comment lines per module, referred to as cl, was separately computed and added as an explanatory variable. (Also, adding blank-lines produces a linearly redundant variable as the number of blank lines is $LoC - SLoC - cl$.) The final mixed-effects regression model is shown in Equation 4.

$$
\begin{aligned}
\textit{defects} = 1.1 &+ qs(12.6 - 0.0761\,SLoC) \\
&- 0.12\,LoC \\
&+ 0.13\,SLoC \\
&+ 0.22\,cl \\
&+ 4.5 \times 10^{-6}\,LoC \times SLoC \\
&+ 6.0 \times 10^{-5}\,LoC \times cl \\
&- 1.3 \times 10^{-4}\,SLoC \times cl
\end{aligned}
\tag{4}
$$

This model's coefficient of determination ($R^2 = 0.714$), indicates that the model explains just over 71% of the variation in the number of *defects*.

While in some ways this model is more complex (*e.g.*, it includes three interaction terms) the interpretation of *qs* is simpler (its coefficient includes only a constant and *SLoC*). Considering *qs* first, the QALP score has the desired effect for modules (files) over a certain size (in this case 166 *SLoC*, the point at which $-0.0761\,SLoC$ is larger than 12.6). In other words, the desired inverse relationship between the QALP score and defect rate is seen when there are more than 166 source lines of code in a file (this is true for 49% of $\mathcal{MP}$'s files). Above this size, a higher QALP score indicates that the module will have fewer defects, whereas a low QALP score indicates there will be a higher number of defects. The slope of this relationship increases as *sloc* increases. Fault prediction for smaller files is less of an issue as inspection of files smaller than 166 *SLoC* is not too demanding. This is particularly true of $\mathcal{MP}$ where each file includes multiple functions.

Similar to the model of Equation 3, in this model as module size increases, this coefficient grows increasingly negative; thus, QALP scores are more valuable for larger modules. The other non-interaction variables (the next three lines of the equation) also indicate that *defects* increase with increased code size. This is best seen by expanding the *LoC* term into *SLoC*, cl, and blank lines. Recombining terms leaves *SLoC* and cl both with positive coefficients (of 0.01 for *SLoC* and 0.1 for cl). It also leaves blank-lines with a negative coefficient, so (presumably up to some limit) white space improves code quality.

As with Mozilla, an inspection of the code was performed, in this case to try and understand why the prediction models were so different for the two programs. The first noticeable attribute of the $\mathcal{MP}$ source is its modularization: each $\mathcal{MP}$ module (*i.e.*, each file) contained many short, well-commented functions. In addition, the comments include those that are *inward-looking* (*i.e.*, they refer to the functionality of the code). These characteristics arose from very strict programming rules adopted by the software group. Each module, independent of its functionality, was built starting from a fixed skeleton into which the engineer inserted code and comments. A team manager directed the project, and a team of three to four senior engineers operated on the framework by specifying the set of libraries at the disposal of the programmers, the skeleton of the modules, and the operating environment. This gave rise to a strict structure for each module in terms of comments, variable declaration (often done with predefined macros), and having the same kind of function in modules of the same type (*e.g.*, those dealing with the GUI all had the same structure independent of the particular data-set they managed). In addition, the type of general comments were specified and were mandatory for principal functions.

In conclusion, for the second of the two programs studied, the QALP score has an inverse correlation with defect rate making it an effective part of a fault-predictor (in the 'comment-lines' model this is when *SLoC* is greater than 166). Furthermore, it improves its effectiveness as the number of source lines of code increases.

4.3 Discussion

This section discusses three points: first an observation made when comparing the models for Mozilla and $\mathcal{MP}$, second, a comparison of the source code for the two programs, and finally the two separate 'kinds' of comments that appear in the code. To begin with, the observation deals with cumulative defects. Previous empirical studies have validated the 80:20 principle, which states that a large majority (around 80%) of problems (*i.e.*, changes or defects) are actually rooted in a small proportion (around 20%) of the code [7, 21, 23]. Using this as a guideline, Bell et al. assess their fault prediction model by ranking files based on the model's prediction and then selecting the top 20% of the ranked files [2]. They report that these files contained 71.1% of the faults. However, this is not a complete picture of the approaches' performance as it does not consider the percentage of faults in the top 20% of files when ranked by *actual* faults. This percentage represents the best performance that a fault predictor can hope for.

Following Bell et al., the top of Figure 5 shows the cumulative percent of faults when modules are ranked using the *predicted* number of faults (gray line) and the *actual* number of faults (black line). The gap between these two curves (shown graphically in the lower part of Figure 5) captures the room for improvement in the prediction. Comparing the charts illustrates the superiority of the model for $\mathcal{MP}$.

Looking at the 20% point in the upper graph for Mozilla, the top 20% of fault-containing modules include 83% of the faults (thus, Mozilla is almost right in-line with the 80:20 rule). Graphically, when sorted on predicted faults, the top 20% of modules include 55% of the faults. Thus, there is significant room for improvement in the prediction.

In comparison, the upper graph for $\mathcal{MP}$ shows the top 20% of fault-containing modules include 62% of the faults

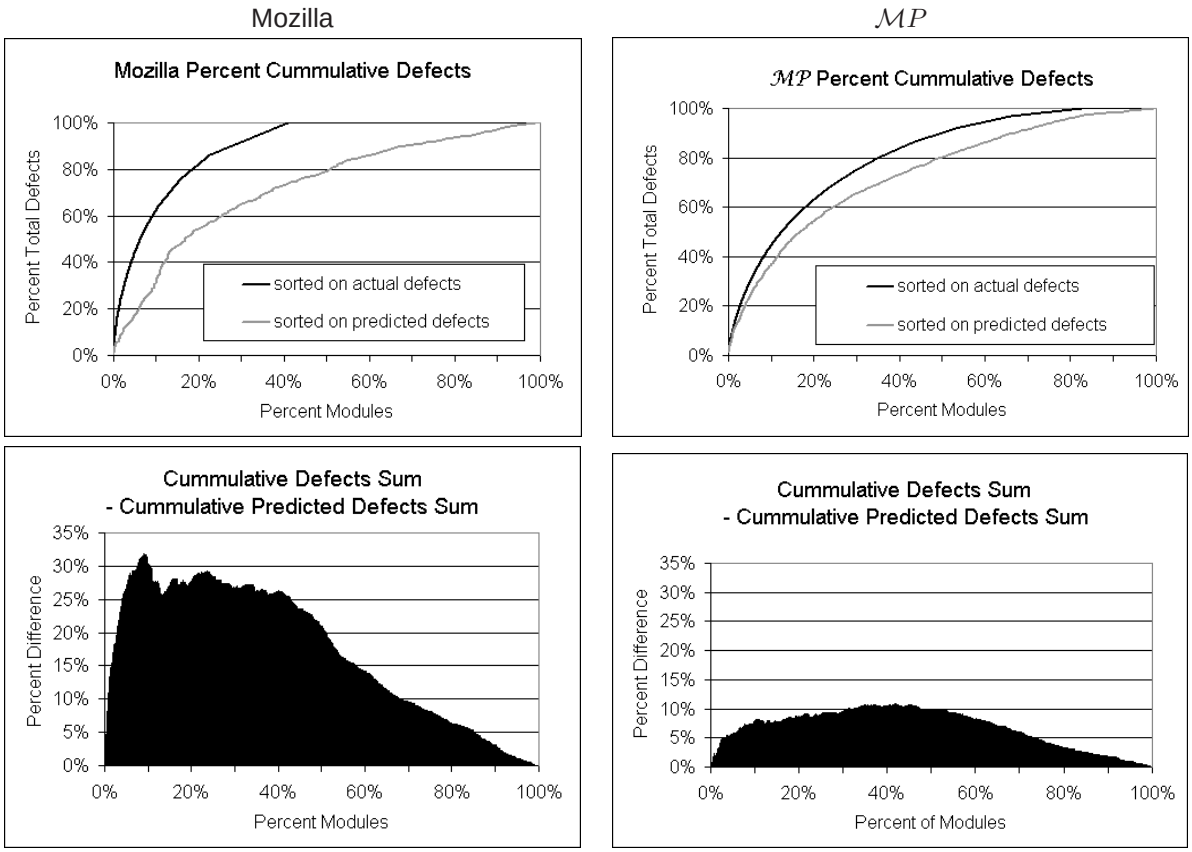

Figure 5. Cumulative Faults

(thus the distribution of faults appears more uniform for $\mathcal{MP}$). When sorted on predicted faults, the top 20% of the modules include only 54% of the faults. While this is significantly less that the 80:20 rules would predict, it is quite good considering that 62% is the best that could be attained (the upper black line). In other words, the model generated for $\mathcal{MP}$ only 'misses' 8% of the possible faults it could find at the 20% point.

Given that Mozilla's model is a comparatively poor predicator of faults, a comparison of the source code for $\mathcal{MP}$ and Mozilla was performed. This comparison revealed that the environment in which $\mathcal{MP}$ was written includes strict style guidelines that encourage a kind of uniformity across all of the source code giving the engineers less freedom in writing code, in writing comments, and in organizing modules. In contrast, the development of open-source software includes "a spectrum of processes from undefined and flexible processes to some extent defined and controlled processes among open-source projects." [10]. Consistent with this Mozilla showed evidence of a diversity of programmers and programming styles.

From the comparison one insight revealed is that QALP scores appear to produce useful information in an environment where the coding style is homogeneous. The reason for this is that extreme variation caused by programmer diversity dwarfs the current predictive ability using QALP scores. Assuming that authorship of code can be ascertained [11], QALP scores may be significant in the Mozilla environment for a particular author, since it can be assumed that a given programmer would use consistent style.

Finally, source inspection also revealed that, in the light of QALP scores, comments can (roughly) be divided into two types: *inward-looking* and *outward-looking*. In the models for $\mathcal{MP}$, the QALP score benefits from the presence of *inward-looking* comments. In contrast, the absence of such comments in Mozilla leads, in part, to an inferior model for Mozilla. Together these observation suggest two things: first, a need for increased inward-looking documentation, and second, the QALP score would benefit from the incorporation of techniques that better assess the value of outward-looking comments.

5 Related Work

This section considers four recent projects in the area of fault prediction. First, Gyimóthy et al. describe the calculation and validation of a collection of object-oriented metrics

for fault-proneness detection [9]. Many of these metrics were originally proposed by Chidamber and Kemerer [3]. They evaluate the metrics by comparing fault predictions against the defects extracted from the Bugzilla database using four assessment methods (*e.g.*, one method used was based on machine learning). The methods all yield similar results. They do note the need for measures not correlated with *LoC*. As the QALP score is not correlated with *LoC* [13], it would be interesting to include it in the predictors generated by Gyimóthy et al.

Second, Koru and Tian show that the top modules in change-count rankings and those with the highest measurement values are different [10]. The authors use change count in preference to defect count as they reported several issues when collecting defect data, such as completeness, consistency in data collection, and problems with mapping defects onto modules. They observe that, at the significance level of $\alpha = 0.01$, the top-change modules are neither the top-measurement modules when identified by ranking nor the top measurement modules when identified by a voting mechanism. Furthermore, using clustering to partition modules into those with similar number of changes, they again observe that the high-change modules are not the modules with the highest measurement values. (The high-change models do have fairly high measurement values.) That structural measures alone do not detect the top-change modules, suggests the need for non-structural measures, such as the information-retrieval based QALP score.

Third, Bell et al. build a fault predictor based on file characteristics that can be objectively assessed: LoC, whether this was the first release in which the file appeared, how many previous releases the file occurred in, how many faults were detected in the file during the previous release, etc. [2]. Based on these characteristics, they build negative binomial regression models to predict files that are likely to contain faults in the next release. Such models are an extension of linear regression designed to handle outcomes for non-negative integers. The advantage of using such a technique is that it allows for some degree of additional variability in fault counts that is not explained by any of the available predicator variables. The use of more sophisticated statistical modelling, as done by Bell et al., partially motivated the use of linear mixed-effects regression models in this study of the QALP score.

Finally, Menzies et al. report that how the attributes are used to build a fault predictor is much more important than which particular attributes are used [16]. They build several predictors using a variety of techniques all starting from the same set of measures. Many of the measures bring similar information to a model and thus different techniques often choose different subsets of the measures, while achieving similar results. The authors note the value that diversity brings to the set of measures. Again, the QALP score, not being correlated with the structural measures, would make an interesting addition.

6 Summary and Future Work

A number of studies have validated the relationship between structural measures and some external attributes associated with problems, such as defectiveness, change-proneness, maintenance difficulty, etc. [2, 7, 9, 10, 16, 19]. One theme noted in many of these studies is the need for more complex, non-structural measures.

This paper presents two case studies of the QALP score, a non-structural IR-based measure that assesses module quality and thus fault proneness. One advantage of the QALP score is that it is applicable during both initial development, and maintenance and evolution. The QALP score is useful in one of two statistical models, which suggests two things: first, the measure has room for improvement, and second information retrieval based measures warrant future study in fault predictors.

Two promising areas of future work include investigating scoring techniques for functions with outward-looking comments and incorporating some measure of 'concept capturing' in the score. In the first area, quality of outward-looking comments might be measured by considering their similarity with the external documentation. The second area is based on the observation of Deissenböck and Pizka that "a reader of a program tries to map the identifiers read to the concepts they may refer to" [5]. Future improvements to the QALP score will attempt to use ideas from machine learning [17] to identify the concepts captured in program identifiers.

7 Acknowledgments

This work is supported by National Science Foundation grant CCR0305330. The authors wish to thank Tibor Gyimóthy's research group for providing the Mozilla fault data and the anonymous referees for providing such extremely helpful comments.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), October 2002.

[2] R. Bell, T. Ostrand, and E. Weyuker. Looking for bugs in all the right places. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*, Portland, MA, July 2006.

[3] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), June 1994.

[4] ML Collard, HH Kagdi, and JI Maletic. An XML-based lightweight C++ fact extractor. *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 134–143, 2003.

[5] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, MO, USA, May 2005. IEEE Computer Society.

[6] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*, Dallas, TX, November 2006.

[7] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8), 2000.

[8] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[9] T. Gyimóthy, R. Ferenc, and I. Siket. Emperical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10), October 2005.

[10] G. Koru and J. Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, 33(8), August 2007.

[11] J. Kothari, M. Shevertalov, E. Stehle, and Spiros Mancoridis. A probabilistic approach to source code authorship identification. In *Proceedings of the 4th International Conference on Information technology: New Generations*, Las Vegas, NV, April 2007.

[12] V. Lavrenko and W.B. Croft. Relevance-based language models. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proceedings on the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, 2001.

[13] D. Lawrie, D. Binkley, and H. Feild. Syntactic identifier conciseness and consistency. In *Proceedings of 2006 IEEE Workshop on Source Code Analysis and Manipulation (SCAM'06)*, Phidelphia, USA, September 2006.

[14] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *14th International Conference on Program Comprehension*, 2006.

[15] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the* 25^{th} *IEEE/ACM International Conference on Software Engineering*, Portland, OR, May 2003.

[16] T. Menzies, J. Greenwald, and A. Fransk. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), January 2007.

[17] T. Mitchell. *Machine learning*. WCB McGraw-Hill, 1997.

[18] C. Morrell, J. Pearson, and L. Brant. Linear transformation of linear mixed effects models. *The American Statistician*, 51, 1997.

[19] J.C. Munson and T.M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5), 1992.

[20] T. Ostrand and E. Weyuker. On the automation of software fault prediction. In *Proceedings of* 1^{st} *Testing: Academic and Industrial Conference (TAIC-PART)*, Windor, UK, August 2006.

[21] A.A. Porter and R.W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2), 1990.

[22] G. Salton and M. McGill. *Introduction to modern information retrieval*. McGraw-Hill Book Company, 1983.

[23] J. Tian and J. Troster. A comparison of measurement and defect characteristics of new and legacy software systems. *Systems and Software*, 44(12), 1998.

[24] C.J. van Rijsbergen. *Information retrieval*. Butterworths, London, second edition, 1979.

[25] G. Verbeke and G. Molenberghs. *Linear mixed models for longitudinal data*. Springer-Verlag, New York, second edition, 2001.

[26] David A. Wheeler. SLOC count user's guide, 2005. http://www.dwheeler.com/sloccount/sloccount.html.