Leveraged Quality Assessment using Information Retrieval Techniques

Dawn J. Lawrie
Loyola College
lawrie@cs.loyola.edu

Henry Feild
Loyola College
hfeild@cs.loyola.edu

David Binkley
Loyola College
binkley@cs.loyola.edu

Keywords: Language Processing, Information Retrieval, Software Quality, Software Assessment

Abstract

The goal of this research is to apply language processing techniques to extend human judgment into situations where obtaining direct human judgment is impractical due to the volume of information that must be considered. On aspect of this is leveraged quality assessments, which can be used to evaluate third-party coded subsystems, to track quality across the versions of a program, to assess the compression effort (and subsequent cost) required to make a change, and to identify parts of a program in need of preventative maintenance.

A description of the QALP tool, its output from just under two million lines of code, and an experiment aimed at evaluating the tool's use in leveraged quality assessment are presented. Statistically significant results from this experiment validate the use of the QALP tool in human leverage quality assessment.

1 Introduction

Assessing any non-trivial aspect of a large piece of software is a difficult task. Unfortunately, for many such tasks, automated techniques have failed to capture human intuition (*e.g.*, as to the quality or complexity of the code). The goal of this research is to leverage techniques from information retrieval (in particular, language processing) into a tool that allows human evaluation of software in the large. Example applications include quality assessment, comprehensibility assessment, and complexity assessment (the complexity of changing a piece of code rather than formal algorithmic complexity).

Historically, Information Retrieval (IR) has been applied to unstructured text (as opposed to the structured information used by database management systems). Recently, IR techniques, which narrow the amount of text a human has to process, "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" [3]. Thus, IR techniques are a natural choice for processing large code bases.

This paper introduces and empirically assesses the application of IR's language processing techniques to quality assessment as implemented in the QALP (Quality Assessment using Language Processing) tool. The QALP tool leverages identifiers and related comments to extract aspects of the program that are representative of the entire program's quality. This facilitates the assessment of large systems where a brute force approach is infeasible.

The QALP approach is quite general in the sense that how quality is defined can vary depending on the extracted quality indicator. For example, one indicator is based on identifiers and thus, for this indicator, code quality is assumed to be correlated with identifier quality. The main focus of the empirical investigation presented in Section 4 is on *function extraction*. For this indicator, code quality is defined in terms of the degree of correspondence of comments and code.

Situations in which such an assessment are useful include the evaluation of third-party coded subsystems and in the selection of code for preventative maintenance. Consider, for example, the case of an organization that hires a third party to code a software subsystem based on a high level design. In addition to integration testing of the delivered subsystem, the organization would benefit from some notion of its quality (and thus, for example, some notion of the future costs to comprehend and maintain the subsystem). While thoroughly assessing the entire subsystem might take as much time as writing it, leveraged quality assessment allows the selection of aspects of the code whose quality is representative of the entire code base.

One motivation for the focus on identifiers is that most of the application-domain knowledge that programmers possess when writing the code is captured by identifiers' mnemonics [3]. Antoniol et al. write "Programmers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar." Thus, an underlying premise of their work and the QALP tool is that programmers use meaningful (high quality) names for code items.

The remainder of the paper includes background information in Section 2, followed by a description of the QALP tool in Section 3. The tool is empirically validated in Section 4. The paper concludes with a discussion of related work in Section 5 and a summary in Section 6.

2 Background

This section provides background on the IR techniques used in the paper including the use of the vector space model, the tf-idf method for term ranking, and the *yari* retrieval engine [17]. Finally, the subject programs and statistical tests used are described.

2.1 Language Processing

The particular IR techniques used in the QALP tool are based on Language Processing (LP). Although LP has historically 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. have independently analyzed comments and variables to (re)establish links between source code and its documentation [3, 22].

In IR the term *document* refers to a cohesive unit of text and is usually the artifact returned as the result of a query. When considering source code, potential "documents" include a class or a function. The collection of documents is processed to construct a corpus. This process, known as indexing, stores information about the terms (usually words) that appear in the documents. Note that the formation of the corpus does not use a predefined vocabulary or grammar.

One method of processing documents is to create a vector space model (VSM) of the documents in the corpus. VSM considers each word w_k as being a separate dimension in an n-dimensional vector space where *similarity* is defined using the cosine of the angle between two vectors. Given that a query can also be expressed as a vector, documents can be ranked according to their similarity to the query. This same measure allows document comparison when used to assess how similar a pair of documents are.

A VSM can be improved by using *stemming* and by employing a *stop-list* to decrease the dimensionality of the vector space. Since IR uses exact matches of words, stemming is generally applied to natural language documents. This eliminates suffixes so that the frequency of a word disregards its particular forms.

A stop-list can be used to omit words that are not thought to be relevant. In English, words such as *"the"* can be eliminated. Since the comments are written in natural English, they are stopped using a standard English stop-list. The code is then stopped using a special stop-list that includes frequently used words that are not unique to the concepts involved in the code. 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.

Once the vector space has been reduced using stemming and stopping, weights are assigned to the words. Of the many alternatives for assigning weights, the standard method, *term frequency-inverse document frequency* (*tf-idf*) is used [26]. It provides a method for weighting the importance of a term (word) 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 (the average number of times the term occurs in a given document) shows how generally important the term is. A high weight using *tf-idf* is thus achieved by a term that occurs much more than the average in a document, but is rare in the entire collection.

The QALP tool uses the search engine Yari, developed by Victor Lavrenko at the Center for Intelligent Information Retrieval [17]. 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 enables the recognition of nouns in natural English.

2.2 Subject Programs

This paper presents empirical data from the 17 programs shown in Figure 1. The figure includes two measures of program size: the first uses the unix utility word-count (`wc`), and the second the utility `sloc_count`, which counts non-comment-non-blank lines of code. As described in Section 3.1, functions are used as documents. The next three columns give the number of functions in each program, the number of functions that include comments either before or within the function, and finally the percentage of functions that include comments. The final column gives a brief description of each program. Note that `gnugo` underwent "objectification" with Version 3.0, which accounts for the large increase in the number of functions.

2.3 Statistical Tests

Several statistical tests are used in the paper. First, when its normality assumption is satisfied, the student's t-test is used to provide statistical confidence when comparing the means of two populations. When comparing means of more than two populations or when normality cannot be ensured, one of four different tests is used. First, the Mann-Whitney test is a nonparametric alternative to the student's t-test for

Program	LOC (wc)	LOC (sloc_count)	function count	functions with comments	percent with comments	Description
go	28,547	25,080	449	339	76%	David Forland's version of go
epwic	8,631	5,245	159	116	73%	Image processor
bc	9,967	6,768	246	234	95%	Calculator
ctags	16,946	13,426	731	255	35%	Emacs and vi tag file generator
diffutils	16,054	10,369	233	187	80%	File compare utilities
flex	20,156	14,455	343	228	66%	Gnu Lex
named	101,827	70,042	2,631	1,687	64%	Bind version 8.2.1
replace	563	512	21	1	5%	String replacement utility
gcc	834,738	588,394	13,555	11,030	81%	The gcc compiler version 2.95
barcode8	4,858	3,198	100	90	90%	A barcode reader
gnugo-1.2	2,857	1,748	31	28	90%	Seven versions of the
gnugo-2.0	26,388	21,721	947	163	17%	Gnu implementation of go
gnugo-2.4	77,977	66,306	2,544	481	19%	
gnugo-2.6	86,558	72,964	2,873	499	17%	
gnugo-3.0	189,658	161,282	7,382	908	12%	
gnugo-3.2	310,449	153,150	12,392	1,200	10%	
gnugo-3.4	186,036	137,885	12,442	1,391	11%	

Figure 1. Subject Programs Studied.

determining if two independent samples come from two different populations. When more than two populations are to be considered, the Kruskal-Wallis test is used. This test determines if at least one sample comes from a different population than the others. When the samples concern the same participant (non-independent) then the Wilcoxon signed-ranks test is used in place of the Mann-Whitney test, and Friedman's test is used in place of the Kruskal-Wallis test. Both these later tests are nonparametric alternatives to an analysis of variance (ANOVA).

Finally, the Kolmogorov-Smirnov test is used to determine whether two underlying probability distributions differ from each other (rather than simply their means as in the aforementioned tests). It does so applying the Mann-Whitney test to the Cumulative Distribution Functions (CDFs) of two data sets. This test determines if the populations from which the samples are taken have different distributions. A CDF is constructed in a nonparametric manner; thus, no default distribution shape is assumed. In the case of a test involving more than two samples, when there is evidence of a difference, the Student-Newman-Keuls (SNK) test is used to identify groups of samples with significant differences.

For both, the parametric and nonparametric tests, two values are reported, the degrees of freedom and a p-score. The p-score can be interpreted as follows: a value less than 0.01 indicates a strong rejection of the null hypothesis, a value less then 0.05 indicates a rejection, and a value less than 0.10 indicates a weak rejection. A p-score greater than 0.10 means that the null hypothesis cannot be rejected; thus, there is no statistical evidence of a difference.

3 QALP

The techniques employed by the QALP tool are designed to allow an engineer to form an opinion as to the quality of the entire system (unlike metrics where quality is typically measured on some ordinal scale). Each technique takes as input source code (and potentially its accompanying documentation) and extracts from the code (and documentation) various quality indicators. An assessor (a software engineer) is then called on to evaluate these indicators. By evaluating the code from various perspectives, the assessor provides multiple judgments of the code's quality. These judgments are then used to form a human-leveraged quality assessment of the system.

Example extraction techniques include function quality assessment, identifier extraction, machine learning, and external documentation correlations. This section focuses on function quality assessment, which is empirically studied in the next section. To provide some indication of the variety of the techniques, identifier extraction is briefly described at the end of this section. Following Takang et al., both of these techniques are based on Brooks' model of program comprehension, where more comments and higher quality identifier names are assumed to produce code that is easier to comprehend [30].

3.1 Function Quality Assessment

The goal of *Function Quality Assessment* is to group functions by relative quality. If successful, functions from the same group will have similar quality. This allows a quality assessment for an entire program to be obtained by sampling a subset of the functions from each group.

Function quality assessment uses cosine similarity to score functions. A corpus is built for each program individ-

ually (and not for all programs together), which means the *idf* value is relative to the particular program, not many programs. This process begins by dividing a program's source code into segments which are then treated as individual documents for the purposes of applying language processing techniques. The exact source code that makes up a segment can vary with language and application. For example, a class is the obvious choice for an object-oriented language. In this paper, the empirical study involves C programs; thus, the obvious choice for a "segment" is a C function.

The QALP tool divides each function into two "documents": One includes only source code and the other only header and inline comments. The comments are stemmed to remove suffixes and stopped using a standard English stop-list. The code is then stopped using the programming language-specific stop-list described in Section 2. The guiding assumption here is that if the code is high quality then the comments give a good description of the code (future work will consider cases where this is not the case). Furthermore, such code uses identifier names derived from the same concepts as described in the comments.

Roughly speaking (good) comments can be divided into two categories. Those that describe the internals of a function for future maintainers and those designed for external users (callers) of the function. The techniques described in this section work better with the former kind of comment. The latter may not include significant overlap with the code and thus will result in lower cosine similarity. Future work with the QALP tool includes correlations with external documentation. This is expected to work well with comments aimed at external users as the goal of such comments is to provide a different level of abstraction than that of the function's implementation. The external documentation will also be used with functions that contain no comments. These approaches will complement the existing techniques.

After dividing each function, the cosine similarity between source code and related comments is computed for each function using tf-idf term weighting [26]. Figure 2 depicts all the scores for commented functions of at least 25 words from the program gcc. (The analysis omits small functions (those having fewer than 25 words), as they tend to have high similarities, but are otherwise unhelpful when generating quality assessments in the large.) Scores are sorted along the x-axis. Points to the left have higher similarity as measured on the y-axis. The empirical data presented in the next section shows that functions with higher cosine similarities receive higher human quality assessments.

A correlation between tool score and quality allows the scores to be used to partition functions such that two functions in the same partition should receive similar assessment if shown to an engineer (hereafter the term "score" refers to the tool's output and the term "rating" refers to values given by programmers including study participants). The partitions are used in leveraged quality assessment where they make it possible to reduce the quantity of code shown to the engineer who will still come away with a sense of the overall quality of the code. As few as one function per partition could be used, but a greater number increases confidence. The QALP tool effectively selects this representative subset of the functions.

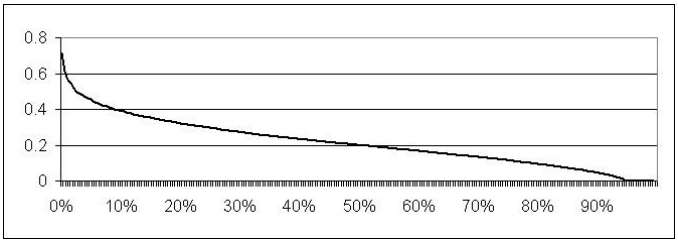

Figure 2. Sorted list of cosine similarities for the 8322 functions of gcc that have more than 25 words. The y-axis shows the cosine similarity score. The x-axis shows the percent of gcc's functions considered.

In certain applications of the QALP approach this selection might consider not the entire program, but rather some subset of its modules. This subset might contain those modules supplied by a third party, or might include the modules identified by another tool as relevant for a particular activity. For example, when performing corrective maintenance an extraction tool might identify those modules relevant to a change. The extracted code could then be assessed by the QALP tool to aid in providing a preliminary estimate of the cost to comprehend and subsequently change the code.

A longitudinal application of the QALP tool supports a quick assessment of the relative quality of several versions of a system. For example, Figure 3 presents QALP tool scores for seven versions of gnugo. The most striking feature, easily visible in the chart, is the three distinct bands that correspond to the three major releases studied. The short diagonal line in the lower left of the chart is Version 1.2. The three gray lines are from 2.X versions, and the top three lines are from 3.X versions. At a minimum this is evidence that commenting of the code occurs mostly at the release of major revisions. The distances between these groups are statistically significant (Kolmogorov-Smirnov, $p = 0.0146$, *d.f.* = 4663) (recall that the Kolmogorov-Smirnov compares distributions). The SNK test applied to all seven versions reports three groups that correspond exactly to the three major versions of gnugo.

3.2 Identifier Extraction

Rilling and Klemola observe, "In computer programs, identifiers represent defined concepts [and] identifier den-

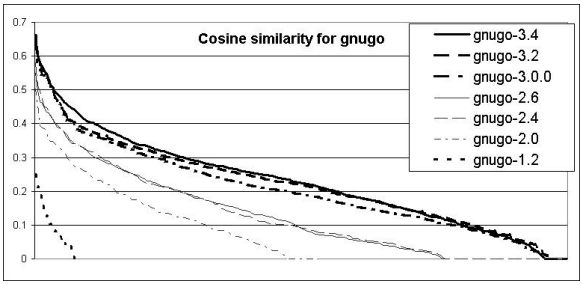

Figure 3. Seven versions of gnugo**. The y-axis shows the tool score. The x-axis shows each function sorted by score.**

sity corresponds to comprehension cost" [25]. Knuth noted that descriptive identifiers are a clear indicator of code quality and comprehensibility [16]. Identifier extraction is based on these observations and the assumption that the clarity of highly used identifiers is of particular importance.

In fairness, quality programs can have poor identifiers (even though it is expected that quality identifiers would accompany quality programs); thus, this information is not a perfect barometer. Extracting identifiers is, however, in keeping with the QALP tool's goal of extracting a variety of indicators that a software engineer can use as evidence in forming an overall assessment.

It is reasonable to ask if IR techniques are really necessary as, for example, a developer could quickly assess identifiers quality by glancing over the source code (identifiers might even be more meaningful in context). While for small to mid-sized codes this is a reasonable approach, one of the goals of the QALP project is the assessment of software in the large. For example, Eclipse 3.0M7 has just over 2MLoC and 94,829 different identifiers (which is around the same number of words as in Oxford Advanced Learner's Dictionary) [10]. An unfortunate engineer might sample the Eclipse source code to get a feel for its quality and end up considering particularly good (or bad) identifiers (even in context) and come away with the wrong impression.

Of course, looking through 2MLoC is a daunting task. By selecting key identifiers, assessment of software in the large becomes possible. The difficult task is defining which identifiers are *key*. In the preliminary study discussed in Section 4.4, this is done based on frequency. Further empirical work will compare alternate definitions of "key," for example, those used in the most files, those declaring common data structures, or even those with high *tf-idf* scores in a module relative to the entire program. Such a study will also consider the pros and cons of considering identifiers out of context.

To be used as a quality indicator, the extracted identifier list is stopped; thus, removing common type names, single character identifiers, library function names, etc. and then sorted based on frequency of occurrence. While the present tool does not perform this step, it would be possible to factor

`replace`		`gnugo 2.0`	
freq	identifier	freq	identifier
62	pat	98	board_size
43	result	86	dragon
31	arg	53	mx
25	junk	52	originj
24	lin	52	origini
9	lastj	19	found_one
9	escjunk	11	GRAY_BORDER
7	dest	10	propogate_worm
7	CLOSURE	10	liberties
6	size	9	strategic_distance_to_white

Figure 4. Frequency sorted identifier lists from `replace` **and** `gnugo`**. In all,** `replace` **has 75 unique identifiers while** `gnugo` **has 134.**

in syntactic context by, for example, separating local variables by context.

For example, Figure 4 includes partial identifier lists from the programs replace and gnugo. The top five most frequent identifiers are shown followed by five of the better identifiers from the top 30 of each list. An informal inspection of the code by the authors reveals that these identifier are indicative of code quality. For example, even out of context, the identifier names board_size and strategic_distance_to_white convey significant conceptual information. In contrast, even the abbreviations used in replace that could be considered good abbreviations, such as pat, carry cognitive overhead [14]; thus, they accompany code that will, for example, be harder to comprehend.

4 Empirical Validation

An empirical validation study of the function quality assessment was conducted. The design of the experiments and results are reported followed by a discussion of threats to the validity and some preliminary results from a related study of identifier usage. These results are important because they validate the use of the QALP tool in assessing software quality, which is a necessary step in performing leveraged quality assessment.

4.1 Experiment Design

The amount of code used in the survey instrument must be large enough to obtain interesting results, but small enough to allow the survey to be completed in a timely manner. Using the cosine similarities as a guide, the following procedure was used to select the study's 18 functions. Starting from a pool of 52 programs containing 58,959 functions and over 2 million lines of code, Step 1 extracted all commented functions with between eight and fifteen unique identifiers. There were 10,686 functions that met this criteria. Such functions have roughly fifteen to forty lines of code, which make them an appropriate length for the survey instrument. From each program, a triple of functions

was chosen such that it included one function whose cosine similarity score was high, one that was in the middle, and one whose score was low. This limited the pool to 28 programs. Triples from six of these programs were selected for the study. The resulting 18 functions contain a total of 692 lines of code.

The experiment was designed with the following three parts. First, basic instructions (*e.g.*, the scale used for the rankings) and definitions such as, in general terms, the aspects of quality (*e.g.*, that functions be single thought cohesive entities, etc.) were provided. These guidelines did not mention comments or hint at any value in connecting comments and the code. Along with a questionnaire used to gather demographic information.

The second and third parts make up the core of the experiment. The second had participants consider the six function triples (shown in random order). They were instructed to first rate their comprehension of each function on a five point scale from "not at all" to "crystal clear." This allowed, for example, the effect of poor understanding on quality scores to be accounted for. The participants then were asked for relative quality ratings of the three functions and to provided a measure of the confidence of their ranking.

The third part of the experiment asked participants to order the six programs based on overall quality as gauged from the three functions taken from each and to provide a measure of their confidence in this ranking. While it was important to collect the data in this order to familiarize participants with the programs before asking that they order them, the opposite order is preferable for presentation; thus, below the data from the final question is considered first.

The target population from which the participants were drawn includes novice through experienced software engineers. In total eighteen participants took part in the study. All participants were students (including pre- and post-baccalaureate) or faculty at Loyola College (a convenience sample) and ranged from the second year of study through post graduate degree holders. Consistent with similar controlled experiments in software engineering [28] experience did not have a statistically significant effect on responses. This is not a surprise, at least to the extent that the experiment can be categorized as software psychology, where no difference would be expected [28]. However, it is important to note that the number of participants is small for such a difference to appear statistically significant. Finally, given the short duration there was no mortality (drop-outs) among the subjects.

4.2 Experiment Results

Average tool scores and participant ratings for the six programs are shown in Figure 5. The average is the average score of the functions used in the study. Informally, there appears to be agreement for most of the programs.

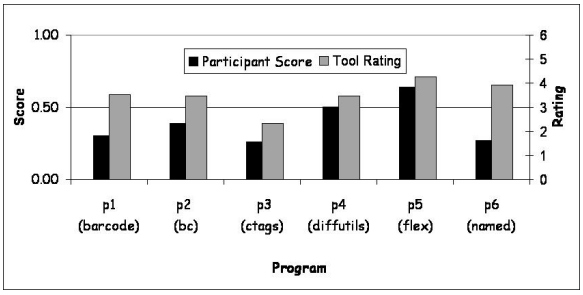

Figure 5. Mean tool scores and participant ratings for each program. (Scores are measured on the left axis and ratings on the right.)

For example ctags it the worst and flex the best using both the participants scores and tool ratings. To formally study relative quality assessment the "*percent precision in pairwise differences*" (similar to the IR metric precision-recall) is used. For each pair of programs the difference in mean score and mean rating is computed. The data is then sorted based on the magnitude of the difference in scores. The average number of agreements in the top k entries yields percent precision through k. The expected result is to find agreement for "larger" differences. As the magnitude of the difference in score approaches zero, agreement is expected to become more random.

Figure 6 shows the sorted scores and ratings for each pair of programs compared. The fourth column includes "yes" iff the tool and the survey agree on the relative ranking of the two programs. If the QALP tool assigned random quality scores, then the probability of agreement would be 50%. If the tool were perfect, then this would be 100%. The penultimate column gives the percent precision. The final column provides the p score comparing the percent precision with a tool that made random quality assignments. As the underlying distribution satisfies the normality test, the student's t-test was used to test the significance of each row.

The expected pattern is present in the data where for differences of at least 0.23 in tool scores, there is 100% agreement. Statistically, for a difference of at least 0.12 in tool score (first underline), the percent precision is significantly different from random. Even down as low as differences of 0.04, there is weak statistical evidence of non-randomness (second underline).

Finally, attributing agreement failures to the programs they involve, the program with the highest score (flex) and the lowest score (ctags) had no agreement failures. Thus, there is consistent agreement on which program is the best and which is the worst. Considering the mean placement of each program, statistically, there is weak evidence that the six means do not come from the same population (Wilcoxon, $p = 0.0537$, *d.f.* = 84). An SNK test identifies two groups: the first includes all the programs except ctags

Programs Compared	Differences **Tool Score**	Survey Rating	Agreement Survey - Tool	Percent Precision	p (Mean > 0.5)
p3 - p5	-0.38	-1.93	yes	100%	-
p5 - p6	0.37	0.33	yes	100%	-
p1 - p5	-0.34	-0.73	yes	100%	-
p2 - p5	-0.26	-0.80	yes	100%	-
p3 - p4	-0.24	-1.13	yes	100%	-
p4 - p6	0.23	0.33	yes	100%	-
p1 - p4	-0.20	0.07	no	86%	0.023
p4 - p5	-0.14	-0.80	yes	88%	0.010
p2 - p3	0.13	1.13	yes	89%	0.004
p2 - p4	**-0.12**	0.00	no	80%	0.026
p2 - p6	0.11	-0.47	no	73%	0.069
p1 - p2	-0.08	0.07	no	67%	0.133
p1 - p3	**0.04**	1.20	yes	69%	0.087
p1 - p6	0.03	-0.40	no	64%	0.151
p3 - p6	-0.01	-1.60	yes	67%	0.104

Figure 6. Tool - Survey Agreement (sorted by magnitude of tool score difference).

and the second includes all the programs except flex; thus, flex with a mean of 4.27 is of higher quality than ctags with a mean of only 2.33. When directly compared, this difference is highly significant (Wilcoxon, $p = 0.003$, *d.f.* = 26). Notice that the QALP tool does not indicate that flex is of *high* quality only that flex is *higher* quality code than ctags.

The data for individual functions is now considered. The question under investigation is do QALP tool scores and the survey ratings assign the same relative order to the functions of each program. To quantify the rankings onto a similar scale as the cosine similarity, the highest ranked function was assigned 1.0 points, the middle function 0.5 points, and the low function 0.0 points. With three functions from each of 6 programs, 18 intra-program comparisons exist. The function comparisons are shown in Figure 7 sorted on the magnitude of tool score difference.

Similar to the program quality comparisons, the difference in ranks was compared to the difference in scores. As shown in Figure 7 the last point at which it is possible to reject the null hypothesis is for differences of at least 0.10 (t-test, $p = 0.049$, *d.f.* = 18). Thus, differences less than 0.10 are not statistically interesting. This point is very close to the score difference of 0.12 found when comparing programs (Figure 6); these findings serve to reinforce each other. Finally, it turns out that the last point for weak statistical difference ($\alpha = 0.10$) is the same as that of statistical difference ($\alpha = 0.05$).

To summarize the two studies, for "sufficient" differences in quality score, the QALP tool's measure of relative quality between functions is consistent with human judgment. This validates the use of the QALP tool in leveraging human quality assessment as the tool can be used to partition functions based on score.

Functions Compared	Differences **Tool Score**	Survey Rating	Agreement Survey - Tool	Percent Precision	p (Mean > 0.5)
5a-5b	-0.46	-0.06	yes	100%	-
4a-4c	0.39	0.31	yes	100%	-
4a-4b	0.33	0.06	yes	100%	-
5b-5c	0.29	0.13	yes	100%	-
1a-1b	-0.20	-0.06	yes	100%	-
5a-5c	-0.18	0.06	no	83%	0.016
6a-6c	-0.17	0.16	no	71%	0.071
1b-1c	0.17	0.03	yes	75%	0.032
2a-2c	-0.16	0.31	no	67%	0.096
6a-6b	**-0.10**	-0.16	yes	70%	0.049
2b-2c	-0.09	0.16	no	64%	0.118
2a-2b	-0.07	0.16	no	58%	0.225
6b-6c	-0.07	0.31	no	54%	0.358
3a-3c	-0.07	0.22	no	50%	0.500
3b-3c	-0.06	0.16	no	47%	0.625
4b-4c	0.06	0.25	yes	50%	0.500
1a-1c	-0.04	-0.03	yes	47%	0.625
3a-3b	-0.002	0.06	no	44%	0.730

Figure 7. Function level agreement (sorted by magnitude of score difference).

4.3 Threats to Validity

In any empirical study, it is important to consider threats to validity (*i.e.*, the degree to which the experiment measures what it claims to measure). There are four types of validity relevant to this research: external validity, internal validity, construct validity, and statistical conclusion validity.

External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized to other organizations or settings. In this experiment, selection bias is possible as the selected functions, programs, and participants may not be representative of those in general; thus, results from the experiment may not apply in the general case. Careful selection of the functions mitigates the impact of their selection. It is possible, but believed unlikely, that programs written for domains not considered (*e.g.*, real-time systems, embedded systems, event-driven systems, or even non-open source programs) may exhibit significantly different behavior. Finally, only a few experienced programmers took part in the experiment. While their data does not appear different from the averages, there are insufficient data for statistically significant conclusions to be drawn. It is possible that repeating the experiment with only advanced software engineers would produce stronger or opposite results. The latter would render the tool of greater use by junior professionals.

Second is the threat to internal validity: the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. The

only significant potential threat to internal validity comes from the three groups that took part in the survey doing so at different times. Thus, it is possible that some participants gained advanced knowledge of the identifiers, functions, or questions. This is believed to be unlikely. Other potential threats to internal validity, for example, history effects, attention effects, and subject maturation [28] are non issues given the short duration of the experiment. As is customary, selection effects were addressed using random selection. Finally, it is possible that exposure to early questions had an effect on responses to later questions. No evidence of this was found in the participants responses.

Construct validity assesses the degree to which the variables used in the study accurately measure the concepts they purport to measure. As human assessment of quality is rather subjective, it is possible that some other aspect of the functions (and identifiers) assessed affected participants ratings'. Indicators such as participants confidence were used to mitigate threats to construct validity. A lesser issue is the threat from potential faults in the QALP Tool. To mitigate this concern, mature IR tools were used and thoroughly tested. This reduces the impact implementation faults may have on the conclusions reached.

Finally, a threat to statistical conclusion validity arises when inappropriate statistical tests are used or when violations of statistical assumptions occur. The statistical tests used were chosen based on past experiments and guidance of those trained in statistics. This serves to reduce the possibility of an inappropriate test being employed.

4.4 Preliminary Results on Identifier Lists

Two motivations for using identifier lists comes from the work of Caprile and Tonella, who observe "identifier names are one of the most important sources of information about program entities" [6] and the work of Deienbck and Pizka, who observe that "research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process." Thus, they conclude that the importance of identifier names is not surprising [10].

A preliminary experiment was run that considered identifier quality. The underlying hypothesis is that a list of frequent identifiers can be a useful piece of evidence in forming an overall quality assessment. In this experiment the top 30 most frequent identifiers were extracted from four programs. Each participant was asked to predict the program's quality based solely on the identifiers. Rather than bias participants with the author's definition of identifier quality, participants were instructed to apply the rules they use when coding themselves.

Statistically, the quality ratings of the four programs are drawn from different populations (Friedman, $p < 0.0001$, *d.f.* = 77). An SNK multiple comparison yields two groups: Group A (p1), which is better than Group B (p2, p3, and p4). In a detailed inspection of the four programs by the authors (performed before the experiment was conducted), p1 had the highest quality, followed by p2, p3, and finally p4. While there is not sufficient data for statistical comparison, it is interesting that this order is consistent with the two groups obtained using only identifier based quality ratings. As identifiers carry significant semantic information, this supports their incorporation as one piece of evidence in the quality assessment of a program.

5 Related Work

This section considers work related to the QALP tool and, where appropriate, highlights the similarities and differences with the QALP approach. First three general applications of IR to source code are considered, and then a progression towards work more closely related work to QALP is considered. This progression first samples projects that consider general quality metrics, and then those that extract information from comments, identifiers, and finally their combination.

Three general areas of IR application to source code emerge from the literature. First, IR techniques have been used to improve, track, and identify the impact of requirement changes. Dag et al. presents an automated similarity analysis of textual requirements using IR techniques [9]. They report that the technique helped engineers identify relationships between requirements, including requirements duplicates and interdependencies. Hayes et al. report success with the related problem of improving requirements tracing based on framing the problem as an information retrieval (IR) problem [13]. Finally, Antoniol et al. propose an IR-based method that given a maintenance request, effectively identifies the set of system components initially affected by the maintenance request [2].

The second area is (re)establishing links between source code and its documentation. Here, Maletic et al. [20] used Latent Semantic Analysis (LSA) to find links between source code and documentation. They worked with comments and identifier names within the source code to extract semantic meaning with respect to the entire input document space [22]. In similar work, Antoniol et al. and De Lucia et al. investigated the use of IR methods to recover traceability links between source code and documentation, and between source code and requirements [3, 18]. Two case studies support the hypothesis that the probabilistic and the vector space IR models are suitable for recovering traceability links between code and documentation.

The QALP tool's use of cosine similarity is similar to the use of cosine similarity in these techniques. While the information gathered is applied to a different use, internally the only significant difference is that the present QALP tools focuses on only internal documentation and the techniques of

Antoniol and Marcus focus on the external documentation. Future work with the QALP tool includes incorporating information from the external documentation.

The third and final area finds commonalities directly in the source code. This is similar to the early work of Maarek [19] on the use of IR to automatically construct software libraries. More recently, Marcus et al. use LSA to identify semantic similarities between source code documents [21]. In similar work Kawaguchi et al. describe an automatic software categorization algorithm to help find similar software systems in software archives [15]. They explore several known approaches including code clones-based similarity metric, decision trees, and latent semantic analysis. Finally, in a related vein, Marcus et al. address the problem of concept location using latent semantic analysis [23]. Two concept locators are presented–one based on user queries and the other on partially automated queries.

Most existing code based (quality assessment) metrics focus on distilling a program down on to an ordinal scale. For example, the object-oriented quality metrics proposed by Chidamber and Kemerer [8] are all integer values. Although they rely more on syntactic structures of C++ code to predict quality, the techniques do associate metrics with (a type of) software quality; empirical investigation of these metrics [4, 5, 11], shows that they predict faults in classes (higher metric values as associated with classes where a high number of faults are found).

Two things distinguish the QALP approach from this prior work. First, existing metrics tend to have a high correlation with the simple metric "lines of code" [12], which the techniques considered herein do not. Second, while it computes numeric scores internally, the focus of the QALP approach is to extract quality indicators from the source code and then allow a programmer to evaluate them thereby leveraging the programmer's intuition and experience in forming a quality assessment.

Representative work that extracts information from program comments is that of Sayyad-Shirabad at el. who propose a technique for the creation of a "light-weight" conceptual knowledge base aimed at helping an apprentice "understand, navigate, and search within the code" [27]. Their process assumes that all the important high level concepts are mentioned in the comments. These concepts are extracted and then revised by a domain expert.

Switching to work that focuses on identifiers, Anquetil and Lethbridge (among others) have observed that there is some controversy over the value of general identifier names [1]. For example, Sneed states that "in many legacy systems, procedures and data are named arbitrarily $\cdots$ programmers often choose to name procedures after their girlfriends or favorite sportsmen" [29]. This pattern was observed by one of the authors at a previous industrial position in the coding of a colleague who was fond of Star Wars. Following Anquetil et al., the QALP tool assumes that software engineers are trying to provide significant names. With the spread of true engineering discipline in the software construction process, this assumption grows increasingly more likely to be satisfied.

An example of a research project that considers both comments and identifiers is the work of Takang et al. [30], which describes testing three hypothesis: (i) commented programs are more understandable than non-commented programs; (ii) programs that contain full identifier names are more understandable than those with abbreviated identifier names; and (iii) the combined effect of comments and identifier names tend to enhance the understandability of a program more than the independent effect of comments or identifier names.

Using a two-way analysis of variance to study the results of objective and subjective information gather from a survey instrument, hypothesis (i) was supported by the objective test scores, and hypothesis (ii) was supported by the subjective scores.

Interestingly, there was no perceived improvement with the combined effect of comments and full identifier names. The authors observe that "This may have been because the comments didn't add a significant amount of information (the program was fairly straight forward without comments)." This can be contrasted with the "real world" code used in the experiment described herein. By providing multiple aspects for an engineer to assess, the QALP tool attempts to provide the best of both worlds as sometimes objective assessments and sometimes subjective assessments are preferable.

6 Summary and Future Challenges

The goal of the QALP tool is to leverage human insight, intuition, and judgment to assess code quality in the large. It does so by using aspects of the system extracted by IR techniques. The QALP tool fills the gap where automated techniques have been found lacking and where direct human assessment is prohibitively expensive. Leveraged quality assessments can be used, for example, to assess the compression effort (and subsequent cost) required to make a change and to identify parts of a program in need of preventative maintenance. The empirical study from Section 4 demonstrates the promise of the QALP approach.

Three promising areas for future work include, investigating scoring techniques for functions without comments. Here it is possible that good internal documentation (primarily through good identifier names) obviates the need for (redundant) commenting. One approach to this end will score a function based on a correlation between identifiers and the external documentation. A second area includes plans to extract other indicators including CVS comments borrowing an idea from Chen et al. who use CVS com-

ments as a source of information for code search [7]. A third area of future work is based on the observation of Deienbck et al. that "a reader of a program tries to map the identifiers read to the concepts they may refer to" [10]. Future QALP research will attempt to use ideas from machine learning [24] to select key components from a program based on higher-level concepts. Parallel to the development of these new techniques, is an ongoing investigation of correlations between leveraged quality assessments and other measures of quality such as bug frequency, reliability, evolvability, robustness, performance, security, correctness, and portability.

7 Acknowledgments

This work is supported by National Science Foundation grant CCR0305330. Chris Morrell and Beth Domholdt provided invaluable statistical assistance. Erin Ptah, Jeri Hanly, and Margaret Daley were extremely helpful in editing the paper.

References

[1] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, November 1998.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance and reengineering. In *Proceedings of 4th European Conference on Software Maintenance*, Zurich, Switzerland, 2003.

[3] 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.

[4] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering*, 22(10):751–761, 1996.

[5] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[6] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[7] A. Chen, E. Chou, J. Wong, A. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the first IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 364–373, Florence, Italy, November 2001.

[8] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[9] J. Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson. A feasibility study of automated natural language requirements analysis in market-driven development. *Requirements Engineering*, 7(1), June 2002.

[10] 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.

[11] R. Ferenc, I. Siket, and T. Gyimthy. Extracting facts from open source software. In *Proceedings of 2004 International Conference on Software Maintenance*, pages 60–69, Chicago Illinois, USA, September 2004. IEEE Computer Society Press, Los Alamitos, California, USA.

[12] T. Gyimthy, R. Ferenc, and I. Siket. Emperical validation of object-oriented metrics on open source software for fault prediction. volume 31, pages 897–911, Washington, DC, October 2005. IEEE Computer Society.

[13] J.H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements traceability via information retrieval. In *Proceedings of 11th IEEE International Requirements Engineering Conference*, Monterey, California, September 2003.

[14] D. Jones. Memory for a short sequence of assignment statements. *C Vu*, 16(6):15–19, December 2004.

[15] S. Kawaguchi, P.K. Garg, M M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *Proceedings of International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003.

[16] D. Knuth. *Selected Papers on Computer Languages*. Stanford, California: Center for the Study of Language and Information (CSLI Lecture Notes, no. 139), 2003.

[17] 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*, pages 365–379, 2001.

[18] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artifact management system with traceability recovery features. In *Proceedings of IEEE International Conference on Software Maintenance*, Chicago, IL, September 2004. IEEE Computer Society Press, Los Alamitos, California, USA.

[19] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8), 1991.

[20] J. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of* 12^{th} *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, Vancouver, British Columbia, November 2000.

[21] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of Automated Software Engineering*, San Diego, CA, November 2001.

[22] 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.

[23] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *IEEE Working Conference on Reverse Engineering*, Delft, The Netherlands, November 2004.

[24] T. Mitchell. *Machine Learning*. WCB McGraw-Hill, 1997.

[25] J. Rilling and T. Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the* 11^{th} *IEEE International Workshop on Program Comprehension*, Portland, Oregon, USA, May 2003.

[26] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1983.

[27] J. Sayyad-Shirabad, T. Lethbridge, and S. Lyon. A little knowledge can go a long way towards program understanding. In *5th International Workshop on Program Comprehension*, pages 111–117, Dearborn, MI, USA, May 1997. IEEE Computer Society.

[28] D. Sjberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N. Liborg, and A. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 19(4):379–389, 1993.

[29] H. Sneed. Object-oriented cobol recycling. In *3rd Working Conference on Reverse Engineering*, pages 169–178. IEEE Computer Society., 1996.

[30] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: An experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.