Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **27**:255–273
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1710

The impact of vocabulary normalization

Dave Binkley*,† and Dawn Lawrie

Computer Science Department, Loyola University Maryland, Baltimore, MD, USA

ABSTRACT

Software development, evolution, and maintenance depend on ever increasing tool support. Recent tools have incorporated increasing analysis of the natural language found in source code, predominately in the identifiers and comments. However, when coders combine abbreviations and acronyms to form multi-word identifiers, they, in essence, invent new vocabulary making the source code's vocabulary differ from that of other software artifacts. This vocabulary mismatch is a potential problem for many techniques imported from information retrieval and natural language processing, which implicitly assume the use of a single common vocabulary. Vocabulary normalization aims to bring the vocabulary of the source in line with that of other artifacts.

A prior small-scale experiment demonstrated the value of vocabulary normalization for C code. A more comprehensive experiment using Java code is presented where normalization fails to bring benefit. To investigate the potential underlying causes, over 20,000 non-dictionary words extracted from the program JabRef were normalized by hand (often requiring significant external information). The experiment, repeated using the hand-normalized identifiers, again found that normalization brought no improvement. In response to this unexpected result, the vocabulary differences between Java and C codes are considered and used to help frame directions for future work. Copyright © 2015 John Wiley & Sons, Ltd.

Received 2 August 2013; Revised 29 December 2014; Accepted 28 January 2015

KEY WORDS: identifier expansion; vocabulary normalization; information retrieval-based tools; case study

1. INTRODUCTION

Tool support plays a significant role in modern software evolution. Effective tools exploit a variety of information to aid a software maintainer. While early tools typically reused the analysis performed by the compiler, recent tools include analysis and information not useful in the compilation process. One class of such tools exploits natural language. This class has the advantage that it can incorporate not only the code but also the artifacts from all phases of software construction and its subsequent evolution.

Tools that exploit the natural language information found in source code include information retrieval (IR) based and natural language processing (NLP) based tools. These tools, which complement traditional static and dynamic analysis tools, have been used to tackle problems that previously required considerable human effort such as (re)establishing links between a program and its documentation [1, 2], assessing program quality [3], developing software metrics [3], and performing concept location [4–7]. This paper takes, as a representative example, a tool that performs one of the most frequent and manually time-consuming software maintenance activities: concept location [8], the activity of identifying the location in the source code of a desired functionality.

*Correspondence to: Dave Binkley, Computer Science Department, Loyola University Maryland, Baltimore, MD, USA.
†E-mail: binkley@cs.loyola.edu

The biggest challenge faced by IR and NLP tools is adapting to the software environment. For example, identifiers capture a significant portion of the domain knowledge found in the code. However, when developing source code, engineers have unprecedented freedom in constructing program identifiers. This leads to considerable invented vocabulary, which is a problem because IR-based and NLP-based techniques implicitly assume that the words (strings) used to express a concept are essentially the same across all the documents of the collection.

Recently, considerable attention has been given to modifying program identifiers in an effort to bring the vocabulary used in the source code in line with other software artifacts (e.g., requirement and design documents, and test plans). When doing so, there is a spectrum of *identifier treatments* that might be applied. This spectrum forms the primary response variable studied in the experiments presented in this paper.

Initially, three points along this spectrum are considered: as a baseline, the unchanged original identifiers, then the conservatively split identifiers [9], and finally, the (aggressively) split and expanded, or *normalized*, identifiers [10, 11]. Conservative splitting undoes the conventions for forming multi-word identifiers by separating identifiers at underscores, transitions from lower to upper case, and between alphabetic and numeric characters. Aggressive splitting goes beyond conservatively splitting by inserting further splits in an identifier to separate underlying concepts not well separated (e.g., conceptlocation is aggressively split into concept location). Finally, *normalization* includes both aggressive splitting and *expansion* of abbreviations and acronyms. Expansion replaces abbreviations and acronyms with dictionary words that capture the meaning of the abbreviation or acronym. As an example, expansion might replace loc with location. Thus, *normalization* differs from splitting in that it goes far beyond splitting and not only separates the parts of an identifier but also expands those parts into one or more dictionary words. As an example, applied to the identifier conceptloc, aggressive splitting separates this identifier into concept loc and then expansion replaces loc with location yielding the normalized identifier concept location.

The key question that guides the project of which this research is a part is, 'Do normalized identifiers impact the performance of downstream tasks?' The specific downstream task considered in this paper is concept location. Thus, the main research question considered in this paper is, 'What impact does Identifier Treatment have on concept location?' With a corpus of two Java programs, Dit *et al.* [12] showed that aggressive splitting brings no significant improvement beyond conservative splitting to concept location. Using a corpus of six Java programs, the experiments presented in Section 5 expand on this work by investigating if vocabulary normalization fairs any better. In considering this question, both *automatic normalization* performed by the tool Normalize [13] and, subsequently, *manual normalization* performed by the authors are considered. The specific contributions of this work are

- an investigation of the impact that automatic normalization (splitting and expansion) has on concept location in Java code and
- a further investigation of manual vocabulary normalization's impact.

To date, this study is the most comprehensive study of the impact the normalization of identifiers has on the downstream task concept location. Furthermore, it improves upon past semi-automatic studies that leverage tool voting (described in Section 8) in investigating the impact of a truly hand-generated oracle of normalized identifiers.

After some background on normalization in Section 2, the investigation begins in Section 3, which presents the paper's two research questions. After describing the experimental design in Section 4, the two research questions are considered in Sections 5 and 6. This is followed by a discussion of the threats to validity, related work, future work, and a conclusion in Sections 7–10.

2. IDENTIFIER NORMALIZATION

To provide context for the study of normalization's impact, some background on the process is provided in this section. To begin with, the notions of *hardwords* and *softwords* are useful in the

DOI: 10.1002/smr

Table I. Normalization study programs.

Program	LoC	SLoC	Unique ids	Oracle ids	Hard words	Soft words
which-2.20	3,670	2,293	487	487	903	1,214
a2ps-4.14	62,347	38,436	4,393	211	459	618

discussion. A *hardword* is a well-separated part of an identifier. Separation occurs at an underscore, at a transition from lower to upper case, and between alphabetic and numeric characters. For example, sponge_bob and spongeBob include the two hardwords sponge and bob. Conservative splitting breaks an identifier into its constituent hardwords.

For some identifiers, splitting into hardwords is sufficient (e.g., when all hardwords are dictionary words); however, for other identifiers, hardword splitting alone is insufficient (e.g., with the identifier spongebob). In this case, further division is required. The resulting strings of characters are referred to as *softwords*. Thus, a softword can be an entire hardword or a sub-string of a hardword. The goal of aggressive splitting algorithms [9, 13–20] is to identify an identifier's constituent softwords. As a final example, consider the identifier hastable_entry. This identifier consists of one separator (an underscore) and thus two hardwords: hastable and entry. The hardword hastable is composed of two softwords, hash and table, while the hardword entry is composed of a single softword.

Other algorithms go beyond splitting to also expand abbreviations and acronyms into dictionary words. Combining these two tasks is referred to as normalization. The experiments reported on herein use the normalization tool Norm [10, 13], which uses the normalization algorithm Normalize. Other similar tools include LINSEN [17], TIDIER [19], and TRIS [20]. Clearly with any such technique, errant normalization (e.g., expanding pos to position rather than positive) can negatively impact results. While not a formal treatment of this issue, the experiments presented in Section 6 shed some light on this issue.

This section reviews Norm's external behavior (e.g., its accuracy) and touches on its internal working and validation. Greater detail can be found in related work [10]. Norm performs two key tasks: splitting and expansion. The aggressive splitting algorithm Gen Test is used, which systematically generates all possible splits of an identifier and then scores them based on a set of features. The features and exact weightings can be found in the work of Lawrie *et al.* [13]. The expansion algorithm, mirroring the process of statistical machine translation [21], exploits co-occurrence data to select the best of several possible expansions from several high scoring splits.

Norm's performance was evaluated using which version 2.20[‡] and a2ps version 4.14[§] [10]. Table I provides statistics on the two programs including two measures of their size (lines of code (LoC) and non-comment non-blank LoC (SLoC) [22]) and statistics related to the extracted identifiers used in the study. All of which's 487 identifiers and a random sample of 211 of a2ps's identifiers were expanded by hand to provide a test oracle.

To avoid diluting the results, the oracle identifiers were further culled by removing 'easy' identifiers from the oracle. These include standard library calls (those with UNIX man pages), names from standard header files (found by searching /usr/include), and language keywords. This removed 152 of which's 487 identifiers and 46 of a2ps's 211 identifiers. The remaining identifiers are not easily understood by someone with a general background in C programming but no domain knowledge. Thus, their expansion should aid a programmer.

The culled oracle was used to evaluate performance. Overall, Norm performs better on which, correctly separating and expanding 65% of the softwords, than it does on a2ps, where it correctly separates and expands 45% of the softwords. In general, a2ps was harder because the vocabulary required to expand abbreviations found in a2ps frequently occurred only in external sources [10].

[‡]http://sourceforge.net/projects/gnuwin32/files/which/2.20/
[§]http://sourceforge.net/projects/gnuwin32/files/a2ps/4.14-1/

DOI: 10.1002/smr

3. RESEARCH QUESTIONS

Past work with Norm has left open the question of its impact on a *down stream* analysis (i.e., the consumer of the normalized source code). Using concept location as a representative downstream analysis, the heart of the experiments presented in Sections 4 and 5 is the impact of the response variable Identifier Treatment (henceforth explanatory and response variables appear in San Serif font). To study the impact of Identifier Treatment, all other conditions (e.g., the stop list employed) are held constant. When applied, each treatment produces a modified version of the source code, which changes the natural language indexed by the search engine. As a baseline, the original identifiers (e.g., max_cnt) are indexed. In contrast, when conservative splitting is applied, the natural language indexed is the separate hardwords of each conservatively split identifier (e.g., max and cnt). Finally, using normalization, the natural language indexed is the separated and expanded words of each identifier (e.g., maximum and count). In all, four values of Identifier Treatment are considered:

1. the original identifiers,
2. the conservatively split identifiers,
3. automatically normalized identifiers, and
4. manually normalized identifiers.

In each case, the response variable is the quality of the ranked list produced by the search engine (the measure used to assess quality is detailed in Section 4.4).

Identifier Treatment's impact is evaluated by considering two research questions. The first considers the first three treatments. Manual expansion is separated out because it is very time consuming. Part of the challenge comes from abbreviations potentially having different expansions in different contexts; thus, each declaration of a variable must be separately examined and expanded. In addition, some expansions require significant research involving information external to the source code. As a result, the first of the two research questions considers a larger corpus of Java programs because the analysis can be automated:

- RQ_1 What impact do the automateable Identifier Treatments have on concept location?

The experiment that investigated RQ_1 revealed a lack of impact, even for normalized identifiers; thus, a second research question was considered that looked at the near-perfect normalization provided by human expanders. Using a corpus of identifiers split and expanded by the authors, RQ_2 investigates whether hand normalization provides any benefit indicating that future research is needed to improve the quality of the automated tools. The second research question focuses on the manual normalization using a smaller corpus:

- RQ_2 What impact does *hand normalization* have on concept location?

4. EXPERIMENTAL DESIGN

Information retrieval-based concept location casts the problem of concept location as an IR search problem. Such searches involve four phases. First, a test collection is identified. Then, the documents of the collection are prepared for indexing, which is followed by the scoring of documents using a retrieval model. Finally, an appropriate evaluation measure must be chosen to interpret the quality of the list of retrieved documents.

4.1. Test collections

In the experiments, two test collections are considered: the *SEMERU collection* [8] and the *Rhino collection* [23]. Each collection includes a set of documents, in this case program methods, a set of queries, which approximate the user's information need, and a set of relevance judgments, which, for each query, identify the documents that should be retrieved. Given a set of documents and a

DOI: 10.1002/smr

Table II. Project statistics.

Project	Number of documents	Size (LoC)	Modification requests	Included queries	Average relevant documents per query	Average length of query (in words)
ArgoUML 0.22	12,448	289 K	91	91	6.3	98.0
Eclipse 3.0	96,309	1.84 M	45	29	2.7	87.2
JabRef 2.6	4,741	117 K	39	39	5.5	110.2
jEdit 4.3	7,118	177 K	150	139	5.9	93.5
muCommander 0.8.5	8,538	171 K	92	87	5.9	83.2
Rhino 1.6R5	3,772	90 K	384	384	9.2	146.0
Total			801	769		

Table III. SEMERU search statistics over all queries.

Project	Relevant methods	Recovered methods
ArgoUML 0.22	701	615
Eclipse 3.0	120	72
JabRef 2.6	280	212
jEdit 4.3	748	546
muCommander 0.8.5	717	518
Rhino 1.6R5	1140	1137

query, an IR search engine's output is a ranked list of documents, ranked based on document score. The relevance judgments can be used to evaluate the quality of this ranked list.

The SEMERU collection consists of five projects [8]: ArgoUML, which builds and maintains UML models; Eclipse, a popular software integrated development environment (IDE); JabRef, which maintains a bibliography (it is primarily designed to manage BibTeX citation libraries); jEdit,[¶] a text editor aimed at programmers that supports a powerful IDE through a sophisticated plug-in architecture; and finally, muCommander, a lightweight, cross-platform, file manager. The Rhino collection consists of single project [23]: Rhino,[‖] which implements the specifications of the European Computer Manufacturers Association Script.[**]

As detailed in Table II, each project includes source code, modification requests (used as proxies for the information needs and thus used to derive the queries), and relevant sets (created by mining the methods changed between the modification request and its closing). The modification requests consist of enhancement requests, feature requests, and patches. Finally, for SEMERU, the queries are formed by concatenating the modification requests' summary and long description, while for Rhino, the two are already combined in the downloadable dataset.

While the queries of the SEMERU collection are based on the modification requests, not all modification requests provide a viable query. This is largely because in the SEMERU collection, relevant documents were identified by recording the names of methods changed between the time that a modification request was submitted and the time of its resolution [8]. Alas, not all of these methods are present in the final indexed version of the program (the version used in the collection). Their absence makes them impossible to retrieve. Table III reports the number relevant methods identified for each project and the subset of these actually present in the indexed collection. When none of the relevant methods for a modification request are in the indexed collection, the modification request has no relevant documents. Such modification requests were excluded from the query set.[††] This occurs when, for example, the methods were present in a previous version but

[¶] http://www.jedit.org/
[‖] http://www.mozilla.org/rhino/
[**] http://www.ecmascript.org/
[††] To support replication, the query identification numbers of the queries used are provided at www.cs.loyola.edu/lawrie/papers/queries-used.txt.

DOI: 10.1002/smr

subsequently deleted from the source code. Thus, in Table II, the number of included queries can be less than the number of modification requests.

For two of the projects, the query sets are further divided based on type: bug or feature. This aligns with the work of Dit *et al.* [12] who used the same two divisions for the same two programs. The numbers of queries and statistics for these two subsets are shown in Table IV.

4.2. Preparing the corpus

Starting with the source code, a three-step process is undertaken to produce the corpus. First, the identifiers are modified in the hope that the modifications will lead to better matches with the language used in the queries. Second, the code is divided into documents at the method level. Finally, a stoplist is applied to remove programming language keywords (e.g., while and if) and traditional English stop words (e.g., is and the).

Three variants of the first step form the three identifier treatments initially investigated in the experiments. The least invasive, Original Source, leaves identifiers unchanged. The middle option, Conservative Split [9], replaces identifiers with their constituent hardwords. Finally, the most invasive treatment, Normalize, replaces identifiers with expanded softwords. To illustrate these three points along the spectrum, the first would leave the identifier avglen as is. Conservative splitting separates this identifier into avglen, while Norm further modifies the identifier to average-length. As a related example, the identifier avglen is left unchanged by all but normalization, where the aggressive splitter produces avglen, which is then expanded to direction.

The same treatments can be applied to the queries. This primarily effects queries that quote source code. As might be expected, when using the original source code, the original (untreated) queries are used. Similarly, when using the conservative split source code, conservative splitting was applied to the queries. As a break from this pattern, for the normalized source code, conservative splitting was applied to the queries. The reason for this is that normalization requires source code context. For example, the expansion of the abbreviation dir in the context of file manipulation code might expand to directory, while in some other context, it might expand to direction. The queries lack the necessary context, and thus, Norm cannot be applied to them. As detailed in Section 6, to investigate if this divergence from the expected pattern negatively impacted the results, a subset of the queries were normalized by hand. Doing so confirmed the inapplicability of an automated approach to normalizing queries. More importantly, using the hand-normalizing queries made no difference in the results, supporting the use of the conservatively split queries with the normalized source code.

4.3. Indexing the collection

Once prepared, the corpus is indexed using the latent semantic indexing (LSI) [24] retrieval model. This facilitates comparison with past experiments [11]. LSI is based on the hypothesis that multiple words may refer to the same concept. In terms of retrieval, the searcher is interested in the concept rather than the particular words used to express that concept. The words that express a particular concept are thus thought of as synonyms (e.g., couch and sofa). The presence of synonyms is one of two challenges in natural language search. The other, homonymy, occurs when the same word (e.g., bank) has different meanings depending on its context (e.g., rivers versus finance).

Table IV. Statistics from queries divided over a program.

Project	Modification requests	Included queries	Average relevant documents per query	Average length of query (in words)
jEdit 4.3 -*Bugs*	86	79	4.0	100.1
jEdit 4.3 -*Features*	64	60	6.3	83.6
Rhino 1.6R5 -*Bugs*	143	143	2.2	135.4
Rhino 1.6R5 -*Features*	241	241	13.4	152.2

DOI: 10.1002/smr

From an implementation perspective, LSI represents documents as vectors with one entry for each unique word in the document. An entry is assigned a weight that reflects the word's frequency in the document. Then words are assigned to topics using singular value decomposition, a process that reduces the dimensionality of the vocabulary from the number of unique words in the project to a specified number of topics. To retrieve a ranked list of documents, queries are projected into the topic (LSI) space. Then the cosine similarity between each document and query is used to rank the documents.

The experiments use the Bell Communications Research implementation of LSI [24]. Apart from two exceptions, the default values for indexing were used. First, the maximum size of a document was increased from 10,000 to 100,000 characters, and second, the number of topics was set to 900. This later value was determined by testing with values from 100 to 1000 in increments of 100 and selecting the best value.

4.4. Evaluation measures

Evaluation of a system's performance provides the measure by which the treatments are compared. The measure used is indirectly based on the IR metrics *precision* and *recall* [25]. Given a set of relevant documents as well as a set of retrieved documents, precision is the number of retrieved-relevant documents divided by the number of retrieved documents. Recall is the number of retrieved-relevant documents divided by the total number of relevant documents.

Although the predominant method for evaluating retrieval effectiveness for ranked lists is *mean average precision* [25], this measure becomes less appropriate when the number of relevant documents for a query is small [26]. In fact, some IR researchers go as far as to exclude queries with fewer than five relevant documents [26]. Such an exclusionary tactic is infeasible in this experiment because over half of queries have fewer than five relevant documents. In such circumstances, a more appropriate measure of performance is the *mean reciprocal rank* (MRR), which is based on the rank of the top-ranked relevant documents. An additional advantage of using MRR is that it does not require the selection of a potentially arbitrary set of retrieved documents, which is necessary to compute precision and recall. Finally, MRR is especially appropriate for tasks such as feature location where finding any one relevant method quickly leads to other relevant parts of the code using other program information (e.g., the call graph).

Mean reciprocal rank averages the reciprocal rank (RR) of the top-ranked relevant document for each query and thus is a measure of precision. RR is calculated using *precision at k*: given a rank k, precision at k is defined as the precision calculated over the set of retrieved documents with a rank of k or less:

$$P(k) = \frac{\text{number of relevant documents in top k}}{k}$$

For RR, k is the highest-ranked relevant document for a query Q, $R_1(Q)$. Thus, the 'number of relevant documents in top k' is always one.

As its name implies, given a set of queries, MRR is the mean of the RR values taken over the set of queries:

$$\text{MRR} = \frac{\sum_{Q \in \text{Querries}} P(R_1(Q))}{|\text{Queries}|}$$

Because performance on individual queries is expected to be highly variable [25], MRR better represents an IR system's overall performance as it smooths over variations and idiosyncrasies possible for any given query.

To minimize the possibility that differences are due to chance factors, statistical analysis is used. Initially, two statistical tests were considered: an analysis of variance (ANOVA) and the Friedman test. The ANOVA is preferred as it compares means. In this case, the mean of the RR values is the standard MRR measure commonly used to compare retrieval results [27]. However, while ANOVA

DOI: 10.1002/smr

is relatively robust with respect to violations of its normality assumption [28], its normality assumption is violated; thus, the Friedman test, a non-parametric alternative to the ANOVA, is used in the analysis. The Friedman test, which tests if k samples come from the same population, is useful where ANOVA's assumption of normality is not acceptable.

5. EXPERIMENTAL ANALYSIS

In research question RQ_1, the impact of Identifier Treatment on concept location is investigated using the two collections: SEMERU and Rhino. Before considering the impact of the three treatments on retrieval, two 'demographic' looks at the data are considered. The first examines the impact on the size of the vocabulary that the treatments have, while the second considers the breadth of each treatment's impact. To begin with, Table V shows the number of unique words indexed by the search engine. This measure of the size of the vocabulary shows that more aggressive changes to the identifiers reduce the size of the overall vocabulary to a greater extent. As can be seen in the table, Conservative Split brings a much larger impact to the size of the vocabulary than Normalize. This is not surprising. First, the vocabulary in the Original Source contains by definition the original identifiers, which was specifically written to contain many unique strings. Undoing the standard composing rules exposes the underlying similarity in the vocabulary make-up. The further reduction with normalization arises from different hardwords representing the same underlying terms. This can come from acronyms whose composing words are already present in the source, identifiers where no visual indicator between terms was included, or when two different abbreviations expand to the same words. Such cases occur less frequently, so the reduction, while present in all programs, is less pronounced.

Second, Table VI explores the proportion of the identifiers impacted by the different treatments. Around half of the identifiers from each program are composed of two or more terms, and thus, conservative splitting modifies them. Norm makes more changes, impacting about two-thirds of the identifiers. The difference between columns 2 and 3 is accounted for by single hardword identifiers untouched by Conservative Split that require normalization (i.e., aggressive splitting and expansion). Comparing this difference to the final column shows that there are some multi-hardword identifiers that are further modified by normalization.

Turning to the impact of Identifier Treatment on concept location, Table VII shows that aggregated over all six programs, the treatments Conservative Split and Normalize outperform Original Source

Table V. Number of unique indexed words by program and identifier treatment.

Program	Original Source	Conservative Split	Normalize
ArgoUML	15,461	9,391	9,112
Eclipse	72,050	28,114	27,145
JabRef	11,207	7,011	6,829
jEdit	12,319	6,904	6,659
muCommander	12,448	7,198	7,093
Rhino	7,324	4,560	4,428

Table VI. Percentage of identifiers modified between different treatments.

Program	From Original Source to Conservative Split	From Original Source to Normalize	From Conservative Split to Normalize
ArgoUML	53.0%	66.5%	22.0%
Eclipse	50.2%	62.1%	20.4%
JabRef	48.6%	61.2%	20.7%
jEdit	49.4%	62.9%	20.1%
muCommander	56.1%	69.3%	19.7%
Rhino	49.5%	62.8%	22.1%

DOI: 10.1002/smr

Table VII. Friedman analysis over all queries in all projects for Original Source, Conservative Split, and Normalize treatments.

	All data	
Identifier Treatment	MRR	Groups
Original Source	0.0960	B
Conservative Split	0.1266	A
Normalize	0.1321	A

by a statistically significant margin. However, although Normalize yields the highest MRR, it is not significantly better than that of Conservative Split.

To explore the data in more detail, Figure 1 shows Identifier Treatment broken out by program. While it is clear from the figure that different programs produce different MRR values, the aspect of the figure relevant to RQ_1 is the 'slope' of each line. In other words, for a given program, what trend emerges over the three treatments. With two exceptions, the MRR values visually show very little variation. Interestingly, the two exceptions show differences in opposite directions. For Rhino Features, there is an improvement in performance mainly from Original Source to the other two treatments. For muCommander, the opposite is true.

In an attempt to illustrate the reasons behind the performance differences for Rhino-Features and muCommander, two examples, one from each program, are considered. While fixation on any given query is unwarranted, it can be instructive to consider individual queries in the hope of gaining an understanding of the circumstances that lead to a particular treatment producing considerably better results. The first example is Rhino Query 125 where Normalize brings benefit over the other two treatments. For this query, Normalize's topped-ranked relevant document is at Rank 7, while Conservative Split manages only 39th and the Original Source a rank of 242. This performance can be attributed to several words from the query appearing only in the normalized source code or appearing more frequently in the normalized source code. One such word is string found in the expanded identifier string_buffer, which is the expansion of the identifier sb. A second example is the expansion of the variable c to character. While less dramatic, the splitting of StringBuffer into string and buffer also increases the occurrences of the query word string. Like Normalize, Conservative Split also splits out the occurrences of string found in StringBuffer and thus also outperforms Original Source.

Query 225 from muCommander is an example where the Original Source performs best. For this query, Original Source produces a relevant document ranked first, while for Conservative Split and Normalize, the first relevant document is ranked 291st and 306th, respectively. This difference occurs because the query explicitly references the class name, FileIcons, which occurs 50 times in the top-ranked document under Original Source. In contrast, when the query is split and used with the Conservative Split and Normalized code, the separated words, File and Icons, are deluded because

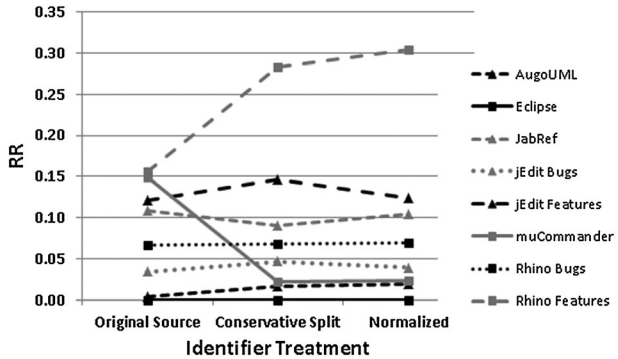

Figure 1. Mean reciprocal rank (RR) values by project for Original Source, Conservative Split, and Normalize treatments.

DOI: 10.1002/smr

Table VIII. Friedman analysis by program for Original Source, Conservative Split, and Normalize treatments.

	ArgoUML		Eclipse		JabRef			jEdit Bugs	
IdentifierTreatment	MRR	Groups	MRR	Groups	MRR	Groups		MRR	Groups
Original Source	0.0034	A	0.0002	A	0.1091	A		0.0344	A
Conservative Split	0.0161	A	0.0001	A	0.0910		B	0.0467	A
Normalize	0.0191	A	0.0002	A	0.1050	A	B	0.0395	A

they occur in many documents. This highlights a downside of the bag-of-words approach, where in this case, it ignores the proximity of file and icons.

Statistically, Table VIII applies the Friedman Analysis out by program. This more detailed analysis reveals that some programs perform best with one treatment, while others perform best when using a different treatment. Relative to RQ_1, the most interesting pattern seen in the table is that statistically Conservative Split and Normalize are never significantly different. In addition, most of the significant differences single out Original Source. For example, the program muCommander has significantly better results with the Original Source, while the two subsets of Rhino have significantly inferior results with Original Source.

Finally, with JabRef Original Source is significantly superior to Conservative Split. The variation in the differences in performance suggests that there are other factors that influence and likely interact with Identifier Treatment. These factors likely are related to characteristics of the program but might also be related to characteristics of the modification requests.

Drilling even deeper, within a program, a few queries do quite well under a particular Identifier Treatment, while others exhibit the opposite behavior. However, MRR smooths out such idiosyncrasies in individual queries; thus, its statistical analysis identifies truly interesting and not just idiosyncratic differences. This is because the reciprocal ranks create the greatest numerical differences when relevant items are ranked near the beginning of the ranked list. Because of this, none of the low-ranked documents are considered outliers because all the low-ranking documents have very similar scores.

Although the mean smooths out query idiosyncrasies, individual queries can be instructive to consider when comparing treatments. For example, Rhino Query 125 considered earlier illustrates the situation where Normalize brings benefit over the other two treatments. Likewise, muCommander Query 225 is an example where the Original Source performs better than the other two.

Finally, a case in which Conservative Split outperforms Normalize and the Original Source is found in jEdit Query 1570554 where Conservative Split ranks a relevant document in the first position versus 100th and 207th for Original Source and Normalize, respectively. The Normalized code more aggressively splits jedit into j and edit to match other camel-cased versions that appear in the code; however, the query includes jedit. Thus, this query word does not appear in the normalized code. When compared with Original Source, other query words such as file occur much more frequently in the Conservative Split (and Normalized) code. This leads to the Original Source ranking this document worse. In fact, it has so little in common with the query that Original Source's top-ranked relevant document is a completely different relevant document.

Summarizing the results for RQ_1, normalization brings no benefit over conservative splitting. The results comparing normalization with the original source are varied and depend on the program (and perhaps the queries) involved. The lack of a strong influence led to research question RQ_2, which aims to understand if Norm's quality is the reason for normalization's lack of significant impact.

6. ORACLE NORMALIZATION

The answer to RQ_1 came as a bit of a surprise and thus calls for investigation. The absence of an influence from vocabulary normalization is either caused by normalization failing to bring value to

DOI: 10.1002/smr

Table VIII. (Continued)

jEdit Features		muCommander		Rhino Bugs		Rhino Features	
MRR	Groups	MRR	Groups	MRR	Groups	MRR	Groups
0.1207	A	0.1491	A	0.0665	B	0.1562	B
0.1463	A	0.0221	B	0.0677	A	0.2831	A
0.1245	A	0.0228	B	0.0694	A	0.3040	A

feature location or by a lack of precision in the normalization algorithm. Research question RQ_2 addresses these two possibilities by considering the influence of near-perfect by-hand normalization: RQ_2 – 'What impact does *hand normalization* have on concept location?'

Study of this question aims to uncover if the errors introduced by imperfect splitting and expansion are interfering with the potential benefit of normalization. To this end, a *normalization oracle* was produced and then used to determine if (near) perfect normalization leads to improved retrieval. This section first describes the process used to generate the normalization oracle and then repeats the analysis presented in Section 5 using the oracle.

The oracle was created by hand. The most ambitious plan would put human eyes on every single instance of all the identifiers in all six programs among the two collections. Even for JabRef, the smallest program among the collections, this would mean checking almost half a million words. This is because different instances of the same identifier can have different meanings in different contexts in the same program. It is not surprising that single-letter and even two-letter hardwords are prone to this. However, three-letter hardwords such as dir can also have multiple meanings (e.g., direction and directory) within the same program.

Given the complexity of the hand normalization task, some simplifying assumptions were necessary to ensure feasibility. First, only a single program was normalized by hand, the smallest JabRef. Second, identifiers were split into *hardwords*. Hardwords not in the dictionary or with fewer than five characters were then examined. The rationale for this focus came from inspecting a sample of hardwords; no dictionary hardword having length five or more ever needed to be expanded into a longer dictionary word. Third, a given hardword was only examined once per source code file. The rationale for this was that the likelihood that a particular hardword has multiple expansions within a file is small, despite being large when considered over the entire program.

The hand normalization process was undertaken by the authors and took approximately 60 hours. After producing all the expansions, some quality control measures were applied such as spell checking the results and examining identifiers that had multiple expansions to ensure consistency. In addition, common acronyms such as pdf and sql were universally not expanded. This choice supports the goal of normalization – producing the same vocabulary as found in other artifacts of a software project. In this case, an acronym such as pdf is much more likely to occur in a bug report than its expanded version, portable document format. To determine how common an acronym was, Wikipedia was used. If an acronym had a Wikipedia page that described the intended meaning in the program, it was considered a common acronym.

Before examining the impact of hand normalization on retrieval, its impact on the source code is considered. Table IX details the differences found between hand-normalized identifiers and the three Identifier Treatments used to explore research question RQ_1. Comparing the different percentages with those presented in Table VI, hand normalization's differences for Original Source and Conservative Split are similar. Despite these percentages being effectively the same, there is a

Table IX. Percentage of identifiers modified between previous treatments and Hand Normalize.

Program	From Original Source to Hand Normalize	From Conservative Split to Hand Normalize	From Normalize to Hand Normalize
JabRef	62.2%	20.1%	23.1%

DOI: 10.1002/smr

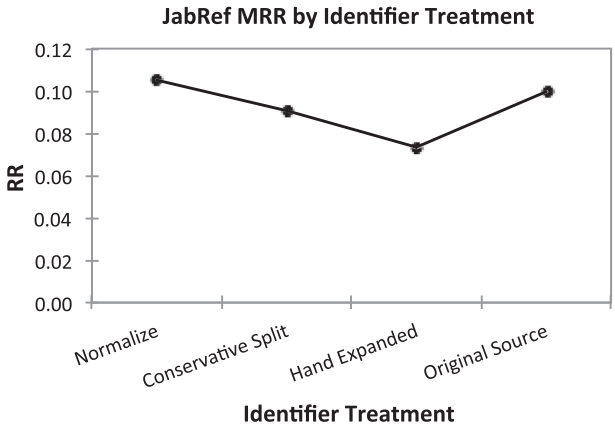

Figure 2. JabRef mean reciprocal rank (MRR) values for Original Source, Conservative Split, Normalize, and Hand-Expanded treatments where hand expansion uses conservatively split queries.

Table X. Friedman analysis for JabRef for Original Source, Conservative Split, Normalize, and Hand-Expanded treatments.

Identifier			
Treatment	MRR	Groups	
Normalize	0.105	A	B
Original Source	0.100	A	
Conservative Split	0.091		B
Hand Expansion	0.073		B

23.1% difference between Normalize and hand normalization. Thus, for JabRef, Norm attains an accuracy of 76.9%.

Although the resulting oracle is not perfect, it provides a clear upper bound on the performance that can be expected from any automatic normalization tool. Using the hand-normalized source code,[‡‡] the experiment from Section 5 was repeated using JabRef. Surprisingly, Figure 2 shows that the hand normalization, labeled Hand Expansion, produced a *lower* MRR score than the other treatments. However, as shown in Table X, the Friedman analysis yields essentially the same results as presented in Table VIII with Hand Expansion joining the 'B' group. Recall that this non-parametric test considers ranks rather than the RR values themselves. This is the reason that Normalize obtains the highest MRR value but is in the group with the two treatments having the lowest MRR values.

Although the overall performance shows no difference, again drilling down, some queries lead to better performance for hand-normalize, while others have better performance with Normalize. Examples of both types are considered. It is easy to believe that hand normalization would improve on the automated tool Norm. A concrete example comes from JabRef Query 1219592 where the query words author, Bibtex, and entry all occur more in the relevant document net.sf.jabref.imports.EndnoteImporter.importEntries(InputStream) when the source has been hand normalized. In the code, Norm fails to expand art to author and b to bibtex_entry (the b occurs in the declaration bibtex_entry b). Both of these variables occur multiple times and thus significantly impact the ranking of this relevant document.

[‡‡]For replication purposes, the hand-normalized version of JabRef can be downloaded from www.cs.loyola.edu/~ lawrie/jabref-2.6-hand-normalize.tar.gz. This download consists of two versions of the source code. In hand-normalize each of the java files consists of the original identifier, the normalized identifier, and the line number where the identifier occurs in the original source code. In src all of the source code is present once the normalized identifiers are substituted for the original ones. Note that this version will not compile because package names were changed during normalization. In addition, each identifier was considered once per file, and thus, there are possible inconsistencies across files.

DOI: 10.1002/smr

In contrast, it is unexpected that Normalize would outperform the hand normalization; thus, it is interesting to consider examples in which this occurs. One such example is JabRef Query 1489454 where Normalize's top-ranked document is at Rank 15, while hand-normalize manages only 105th. The difference comes from hand normalization replacing get_runtime().exec(cmd) with get_runtime().execute(command). In this case, the query includes mention of the program cmd.exe. While this is not the 'cmd' found in the code, its serendipitous matching leads to Normalize outperforming the hand-normalized code.

A second example is Jabref Query 1601651 where hand normalization leaves intact a compound word that Norm split: Norm includes sub-string, while the hand-normalized code includes substring. Given such examples are infrequent, it is no surprise that statistically, there is no performance difference between Normalize and hand-normalize. Such examples seem more appropriately attributed to random chance than a true indications of a difference between the two.

Stepping up a level, one reason that hand-normalize is outperformed by Normalize over all queries is seen when looking at individual query's contribution to the two MRR values. It turns out that the bulk of the difference comes from three queries where a relevant document, which had been ranked at position two, fell. This has a notable impact on the mean RR value but little statistical impact because it only impacts three queries. Overall, half of the queries show no change in score. Of the remainder, about the same number see improved performance as see degradation. Anecdotally, examining one query, the string edtr was hand expanded to editor. Although identifiers are expanded, a key relevant document included this abbreviation in a string that did not get expanded.

As mentioned previously, when the original source is used, the queries are used unmodified. For Conservative Split and Normalize, the queries are conservatively split. It is possible that the vocabulary in the queries also needs to be normalized, so for JabRef's queries, all non-dictionary words were examined and normalized by hand to be consistent with the Hand Expansion. Opportunities to normalize were fewer than in the code. While hand normalization leads to 62% of the queries having at least one word modified, very few words were in fact changed as only 3.5% of the query words were non-dictionary words. A majority of the modifications corrected spelling errors in the queries. The analysis was then repeated with Normalize and Hand Expansion using the normalized queries. Table XI shows the results where hand normalization of the queries actually *lowers* the MRR for both Normalize and Hand Expansion, although these differences are not statistically significant. The important point is that the results support those presented in Table X; reinforcing the lack of value normalization brings Java code.

In summary for research question RQ_2, even under the best case scenario of a tool-performing expansion as well as a human, normalization does not improve IR-based concept location of the Java code considered. Thus, improved normalization tools appear unlikely as a fruitful direction for future investigation at least in the context of Java code. This result reinforces the similar conclusion reached by Dit *et al.* [12] in their evaluation using semi-automatic normalization.

7. THREATS TO VALIDITY

As with any empirical experiment, there are several standard threats to the validity of this analysis. Examples include the programs not being representative and the misinterpretation of the statistical models. A threat to the creation of the collections concerns the set of relevant documents, which

Table XI. Friedman analysis of JabRef using the hand-normalized queries.

Identifier			
Treatment	MRR	Groups	
Original Source	0.100	A	
Conservative Split	0.091	A	B
Normalize	0.078		B
Hand Expansion	0.067		B

DOI: 10.1002/smr

was automatically mined by identifying changed documents between the time a modification request was created and the time it was marked completed. Clearly, there are other reasons that a method might have been changed. On the other hand, of the set of methods that would help an engineer locate a concept, not all will require a change. Thus, mining changed methods both under and over estimates the true relevant set.

Concerning the experimental set up, there is a threat of algorithmic bias. Alternatives to Norm such as LINSEN [17], TIDIER [19], or TRIS [20] might lead to different results. However, the hand normalization results from Section 6 make this unlikely. Finally, this concern does not apply to conservative splitting because it is a well-defined process [9]. A related threat to external validity comes from the impact of different preprocessing steps that might influence the conclusions. While different preprocessing options should definitively change the MRR scores, it is less clear that they would change the trends seen. The outcome is more likely to mirror that pattern seen in Figure 3 where, as discussed as part of future work, an alternate retrieval model to LSI was considered.

Turning to the hand normalization, the process of hand normalization could have introduced threats to validity because English words of five or more letters were not examined. It is possible that some of these words required expansion. In addition, while rare, occasionally a single hardword within a file would expand to more than one concept within the file. Therefore, the hand-normalized code is not 100% accurate. In general, attaining 100% accuracy would require the original programmers to ensure the intended meaning of each identifier was accurately captured by the expansion.

Finally, it is possible that concept location is not a representative task for assessing how IR-based tools will respond to normalization. However, given that this task requires very little reformulation to transform it into an IR task, it should be representative of other tasks that can be translated into an IR task with comparative ease.

8. RELATED WORK

There are a number of different avenues of related work. To begin with, the experiments involve concept location. While there is too great a wealth of literature on concept location (not to mention related tasks such as bug location) to do it justice here, Dit *et al.* provide an excellent systematic survey of this work [8]. The remainder of this section first considers work that addresses a question closely related to the research questions investigated herein: the impact of improved splitting on concept location. This is followed by more general work on the problem of identifier normalization. Finally and further afield, other possible retrieval models for indexing corpora are examined.

Dit *et al.* present an experiment that mirrors in intent the work presented herein [12]. They consider three splitting strategies: conservative, the more aggressive splitter Samurai [9], and semi-automatic normalization (termed 'perfect splitting'). The semi-automatic process manually splits identifiers

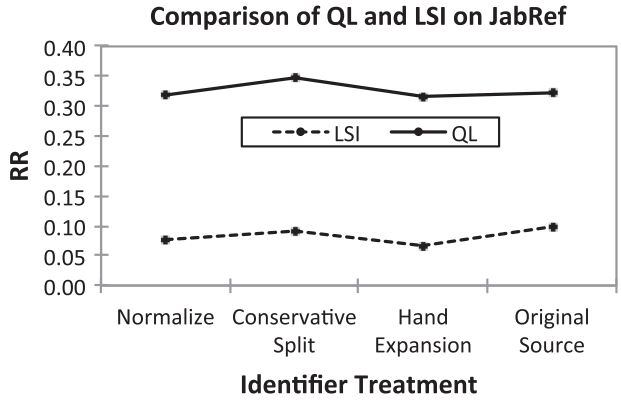

Figure 3. JabRef mean reciprocal rank (RR) values for Original Source, Conservative Split, Normalize, and Hand-Expanded treatments, using the retrieval models latent semantic indexing (LSI) and query likelihood (QL) on hand-expanded queries.

DOI: 10.1002/smr

where three tools (conservative, Samurai, and TIDIER) disagreed. The main research question addressed is 'Does access to semi-automatic normalization improve concept location?'

This question is addressed using two concept location techniques, one based on IR and the other on a combination of IR and dynamic analysis. The experiments use two Java systems, Rhino and jEdit. Overall, none of the treatments lead to a significant difference except in one case where, in the absence of dynamic information, using the perfect splittings proves better than the use of conservative splitting.

In more detail, the exception was with the Java program Rhino where using the conservative split identifiers produces a median rank of 23 and an average rank of 86, while using the oracle-split identifiers produces a median of 20 and, interestingly, the same average of 86. Dit *et al.* also report finding instances where the 'perfect splitting technique can have negative impact on feature location'.

In comparison, the results presented herein are similar, although the Identifier Treatment only overlap with conservative splitting and a manual approach to normalization. Given the cost of manually splitting and manually expanding identifiers, neither study considers a large sample size. However, both lead to a similar observation, which Dit *et al.* state as 'our findings outline potential benefits of putting additional research efforts into defining more sophisticated source code [splitting techniques]' [12]. Given the results from Section 5, when it comes to vocabulary normalization, this additional research should consider languages different from Java.

Others have observed the need for normalization. For example, Corazza *et al.* note that the effectiveness of IR-based techniques to software maintenance and evolution problems worsens if programmers introduce abbreviations [17]. Their solution, which runs in linear time with respect to the size of the dictionary, employs a graph model that takes advantage of an approximate string-matching technique. The solution exploits a number of different dictionaries, referring to increasingly broader contexts. Empirical evidence presented by Corazza *et al.* demonstrates that their approach improves upon techniques that came before it. They study 24 open source systems including both C/C++ and Java. Interestingly, their most dramatic improvements are seen with C/C++ codes.

In recent work, Guerrouj *et al.* [20] describe a tree-based approach that builds arborescence trees from dictionaries derived from the application and external sources. Once the trees are created, identifiers are split and expanded in quadratic time in terms of identifier length based on an auxiliary graph that uses a cost function to find the optimal split and expansion. This alternative approach to vocabulary normalization is very efficient in terms of time and has been rigorously analyzed. Again consistent with the work of Corazza *et al.*, they conclude that their approach outperforms camel casing for C codes but not Java.

Turning, more broadly, to the use of IR techniques to address software engineering problems, one of the key decisions that must be made is which retrieval model to use to index the corpus. While LSI is used in this study, other retrieval models have been proposed. In general, retrieval is accomplished through a process that scores documents with respect to a query. This score can be a measure of similarity, but it does not have to be. As is carried out in this paper, the majority of studies that apply IR techniques to software maintenance problems have involved concept location and LSI [6, 29, 30]. As a consequence, much effort has gone into improving LSI through, for example, its combination with a PageRank-inspired algorithm that relies on dynamic traces [30].

Other retrieval models have been explored. Zhao *et al.* [5] looked at the vector space model. Cleary and Exton [31] compared LSI with a query likelihood model. In addition, latent Dirichlet allocation (LDA) has been applied to feature location in works such as Nguyen *et al.* [32] and Lukins *et al.* [33], where it performs favorably when compared with LSI. Rao and Kak [34] performed a comparison of retrieval models on iBUGS. They conclude that sophisticated models like LDA and LSI do not outperform simpler models like the query likelihood model or vector space model for IR-based bug localization on large software systems. Finally, Binkley *et al.* [35] present a comprehensive study comparing all of these retrieval models. In this study, the query likelihood models are the top performers.

9. DIRECTION FOR FUTURE WORK

This section considers four avenues of future work suggested by the experiments. The first area considers the queries because they can have a big impact on the results. Future work might turn to analysis of the queries and techniques designed to improve them [36, 37].

DOI: 10.1002/smr

Second, while this study focuses on the influence of Identifier Treatment, the data does suggest several other interesting future studies. For example, a search for a correlation between values and size. This pattern is hinted at by Figure 1 where the MRR values are inversely proportional to program size. For example, Eclipse is the largest program and has the smallest scores.

A third avenue comes from the consideration of retrieval models other than LSI. For example, Figure 3 compares LSI with the query likelihood model [38] with Dirichlet smoothing [39]. Two observations are evident: the trend (slope of the lines) is the same indicating that Norm has no significant impact; however, the distance between the two lines is substantial, indicating that Retrieval Model has a greater impact than Identifier Treatment. Therefore, the development of retrieval models explicitly designed to exploit the nature of source code has the potential to provide further improvements.

Fourth, a case study using NCSA Mosaic [6] found that normalization provided a vast improvement. The question is 'Why?' One of the differences between Mosaic and the programs used in the experiments is the programming language used. Rather than Java, Mosaic is written in C. It is possible that normalization's impact varies with programming language, program age, or anywhere where the naming conventions differ from those of Java.

As a preliminary investigation into the differences between the natural language found in Java and C codes, an investigation of the identifiers found in a corpus of C programs was compared with those from a Java corpus to see if there is any statistical differences in the way identifiers are built with respect to conventional rules for composing multi-word identifiers. There has been a general trend toward improved identifier quality over the past 40 years [40]. One implication of this is that Java programs should have higher-quality identifiers given that Java is comparatively the 'new kid on the block.' In addition, from Java's inception, its designers have advocated a particular identifier style, which encourages identifiers built from dictionary words. Examples of this pattern are seen in the standard Java libraries.

While the true measure of the vocabulary used is in the hardwords that make up program identifiers, an initial look at the identifiers considered helps provide context for the deeper study. Five C programs[§§] and the five Java programs of the SEMERU collection are considered. They include a total of 1,391,835 C identifiers and 9,159,322 Java identifiers. While considerably more Java identifiers are involved, over one million instances in each programming language provides a sufficient sample from each language.

The identifier's constituent hardwords more accurately capture the diversity of the language found in the identifiers. In support of the general higher quality of Java identifiers, the 9,159,322 Java identifiers are made up of 11,621,792 hardwords, while the 1,391,835 C identifiers are made up of only 1,556,942 hardwords. When comparing the number of identifiers with the number of hardwords, Java increases by 27%, while C only increases by 12%; thus, Java identifiers are clearly built from a larger number of easily separable parts.

The potential impact of normalization can be initially assessed by considering the ratio of dictionary to non-dictionary hardwords found in C and Java identifiers. In total, 92.5% of the Java hardwords (10,745,685 of 11,621,792) are dictionary words. Because the meaning of most dictionary words is straight forward, this means that only 7.5% of the vocabulary from the Java codes potentially requires decoding. For C, 83.5% of the hardwords (1,300,052 of 1,556,942) are dictionary words. Thus, the percentage of non-dictionary words for Java, 7.5%, is less than half C's 16.5%.

Statistically, a z-test for two proportions (not to be confused with the z-test for comparing means) finds the proportion of dictionary words in the Java codes statistically more than the proportion in C code ($p < 0.0001$). Thus, there is statistical evidence that normalization has greater opportunity in C code. This is a result that confirms the similar observation made by Guerrouj *et al.* [19].

However, the analysis uses Ispell (Version 3.1.20) as the arbiter of which hardwords are 'in the dictionary'. The use of a spell checker not specifically designed for source code raises two concerns. First, some dictionary words, for example, cat and mess, are likely to be abbreviations (in this case

[§§]Mosaic (http://download.cnet.com/NCSA-Mosaic/3000-2356) was supplementedwith gnugo version 3.4 (http://www.gnu.org/software/gnugo/devel.html), LEDA version3.0(http://www.cs.sunysb.edu/~algorith/implement/LEDA/implement.shtml), SNNS version4.2 (http://www.ra.cs.uni-tuebingen.de/downloads/SNNS/), and which version 2.20.

DOI: 10.1002/smr

of concatenate and mesage) when they appear in source code. Second, while not in the dictionary, 'words' such as 'pdf' and 'iteratable' are well known to programmers who are unlikely to value their normalization.

Considering first the dictionary words that should be expanded, the availability of an oracle for the expansion of the hardwords in the programs JabRef (described in Section 6) and which [13] makes it possible to refine the proportion of Java and C dictionary words in need of expansion. JabRef has 444,540 hardwords found in the dictionary. Of these, 21,713 have an expansion in the oracle; thus, the 'error rate' for JabRef's in-dictionary hardwords is 4.7%. For which, the percentage is similar; 3.7% (40 of the 1083 dictionary hardwords were expanded in the oracle).

Next consider words that were not found in the dictionary but should not be expanded because they are part of the solution domain vocabulary. The two oracles again allow an inspection of the error rate by counting the number of non-dictionary words that are not expanded in the oracle. For JabRef, 23%, 11,092 of the 47,797 non-dictionary hardwords, are not expanded and thus are taken as solution domain vocabulary. For which, only 8.3%, 16 of the 192 non-dictionary hardwords, are not expanded and thus are taken as solution domain vocabulary.

Based on the two oracle expansion sets, the dictionary words that were wrongly classified and the non-dictionary words that were wrongly classified can be assigned to the correct set. The z-test is then able to determine if the misclassifications has any impact on the overall conclusion. A z-test for two proportions again finds the proportion of dictionary words in Java code statistically more than the proportion in C code ($p < 0.0001$).

Finally, JabRef turns out to be a good choice from the programs of the SEMERU collection to consider in Section 6. The proportion of non-dictionary words is slightly higher in JabRef than in the SEMERU collection as a whole.

That JabRef has slightly more opportunity for expansion, given its slightly higher proportion of non-dictionary words, makes it a good choice for the analysis described in Section 6 because of the greater potential that normalize will show a difference. In summary, the natural language vocabulary found in C and Java programs differs. In particular, C programs include more opportunities for normalization. If Norm can exploit these opportunities, then it might provide a significant improvement worthy of future research effort. Further support for normalization bringing value to C code is found in the related work of Corazza *et al.* [17] and Guerrouj *et al.* [20].

10. CONCLUSION

The argument in favor of vocabulary normalization is compelling, where a common vocabulary would improve retrieval. However, while IR researchers have begun to address the challenging task of multi-language retrieval, most existing techniques assume all documents share a single common vocabulary. Surprisingly, the data stand in stark contrast to this compelling argument. The experiments presented in this paper use both tool-performed normalization, which while not perfect is correct 75% of the time, and manual normalization, which while not perfect, can be expected to be correct close to 100% of the time. For both kinds of normalizations, there is no statistical evidence that normalization is helpful.

It is possible that the argument in favor of vocabulary normalization is supported when using tasks other than concept location or languages other than Java. However, as this lack of impact has been seen by others, it may be worth undertaking more fundamental research in the understanding and designing IR-based algorithms designed specifically for source code.

ACKNOWLEDGEMENTS

Support for this work was provided by NSF grant CCF 0916081. Any opinions, findings, conclusions, or recommendations presented are only those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering Oct* 2002; **28**(10):970–983.

DOI: 10.1002/smr

2. Marcus A, Maletic J. Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings of the 25th IEEE/ACM International Conference on Software Engineering*, Portland, OR, 2003.
3. Mili H, Ah-ki E, Godin R, Mcheick H. An experiment in software component retrieval. *Information and Software Technology* 2003; **45**:633–649.
4. Shepherd D, Fry Z, Hill E, Pollock L, Vijay-Shanker K. Using natural language program analysis to locate and understand action-oriented concerns. International Conference on Aspect Oriented Software Development: Vancouver, British Columbia, 2007.
5. Zhao W, Zhang L. SNIAFL: towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodology* 2006; **15**(2):195–226.
6. Marcus A, Sergeyev A, Rajlich V, Maletic J. An Information Retrieval Approach to Concept Location in Source Code. IEEE Working Conference on Reverse Engineering: Delft, The Netherlands, 2004.
7. De Lucia A, Penta MD, Oliveto R. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering* 2011; **37**(2):205–227.
8. Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution* 2011; **23**(7):53–95.
9. Enslen E, Hill E, Pollock L, Vijay-Shanker K. Mining source code to automatically split identifiers for software analysis. *Proceedings of the 6th Mining Software Repositories*, 2009.
10. Lawrie D, Binkley D. Expanding identifiers to normalizing source code vocabulary. *ICSM '11*: *Proceedings of the 27th IEEE International Conference on Software Maintenance*, 2011.
11. Binkley D, Lawrie D, Uehlinger C. Vocabulary normalization improves ir-based concept location. *Software Maintenance* (*ICSM*), *2012 28th IEEE International Conference on*, 2012.
12. Dit B, Guerrouj L, Poshyvanyk D, Antoniol G. Can Better Identifier Splitting Techniques Help Feature Location? : International Conference on Program Comprehension, 2011.
13. Lawrie D, Binkley D, Morrell C. Normalizing Source Code Vocabulary. International Working Conference on Reverse: Engineering, 2010.
14. Hill E, Binkley D, Lawrie D, Pollock L, Vijay-Shanker K. An empirical study of identifier splitting techniques. *Technical Report TRLoyola928*, Loyola University 2011.
15. Madani N, Guerrouj L, Penta MD, Gueheneuc Y, Antoniol G. Recognizing words from source code identifiers using speech recognition techniques. *Proceedings of the 14th European Conference on Software Maintenance and Reengineering* (*CSMR*), IEEE, 2010.
16. Butler S, Wermelinger M, Yu Y, Sharp H. Improving the tokenisation of identifier names. *Proceedings of the 25th European conference on Object-oriented programming*, Springer-Verlag, 2011; 130–154. URL http://dl.acm.org/citation.cfm?id=2032497.2032507.
17. Corazza A, Martino SD, Maggio V. Linsen: an approach to split identifiers and expand abbreviations with linear complexity. *IEEE International Conference on Software Maintenance*, ICSM '12, IEEE Computer Society: Washington, DC, USA, 2012.
18. Feild H, Binkley D, Lawrie D. An empirical comparison of techniques for extracting concept abbreviations from identifiers. Proceedings of IASTED International Conference on Software Engineering and Applications, Dallas, TX, 2006.
19. Guerrouj L, Penta MD, Antoniol G, Guéhèneuc Y. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice* 2011. DOI:10.1002/smr.539.
20. Guerrouj L, Galinier P, Guéhéneuc Y, Antoniol G, Di Penta M Tris: a fast and accurate identifiers splitting and expansion algorithm *Proceedings of the Working Conference on Reverse Engineering*, 2012.
21. Gao J, Nie JY, He H, Chen W, Zhou M. Proceedings of the 26th ACM SIGIR Conference, 2002.
22. Wheeler D. SLOC count user's guide 2005. Available from: http//www.dwheeler.com/sloccount/sloccount.html (accessed on 27 February 2015).
23. Eaddy M, Zimmerman T, Sherwood K, Garg V, Murphy G, Nagappan N, Aho A. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering* 2008; **34**(4):497–515.
24. Deerwester S, Dumais ST, Furn GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 1990; **41**(6):391–407.
25. Manning C, Raghavan P, Schutze H. Introduction to Information Retrieval. Cambridge University Press, 2008.
26. Wei X, Croft WB. LDA-based document models for ad-hoc retrieval Proceedings of the Twenty-Ninth Internation ACM Conference on Research and Development in Information Retrieval, Seattle, WA, 2006.
27. Voorhees EM, Tice DM. Building a question answering test collection. *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, 2000.
28. Kirk RE. Experimental Design: Procedures for the Behavioral Sciences, 3rd edn. Pacific Grove, CA, USA: Brooks/Cole, 1995.
29. Poshyvanyk D, Guéhéneuc Y, Marcus A, Antoniol G, Rajlich V. Combining probabilistic ranking and latent semantic indexing for feature identification. *14th International Conference on Program Comprehension*, 2006.
30. Revelle M, Dit B, Poshyvanyk D. Using data fusion and web mining to support feature location in software. *Proceedings of the 18th International Conference on Program Comprehension*, 2010.
31. Cleary B, Exton C. Assisting concept location in software comprehension. *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group*, 2007.
32. Nguyen A, Nguyen T, Al-Kofahi J, Nguyen H, Nguyen T. A topic-based approach for narrowing the search space of buggy files from a bug report. *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

33. Lukins S, Kraft N, Etzkorn L. Source code retrieval for bug localization using latent Dirichlet allocation. *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008.
34. Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011.
35. Binkley D, Lawrie D, Uehlinger C, Heinz D. Enabling improved IR-based feature location. *Journal of Systems and Software* 2015; **101**(0): 30–42, DOI: 10.1016/j.jss.2014.11.013. Available from: http://www.sciencedirect.com/science/article/pii/S01641212140 02428 (accessed on 27 February 2015).
36. Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T. Automatic query reformulations for text retrieval in software engineering. *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, ICSE 2013, 2113.
37. Balasubramanian N, Kumaran G, Carvalho V. Exploring reductions for long web queries. *Proceedings of the 33rd ACM SIGIR Conference*, 2010.
38. Ponte J, Croft WB. A language approach to information retrieval. *Proceedings of the 21st ACM SIGIR Conference*, 1998.
39. Zhai C, Lafferty J. A study of smoothing methods for language models applied to ad hoc information retrieval. *Proceedings of the 24th ACM SIGIR Conference*, 2001.
40. Lawrie D, Feild H, Binkley D. Quantifying identifier quality: an analysis of trends. *Journal of Empirical Software Engineering* 2007; **12**(4):359–388.