Expanding Identifiers to Normalize Source Code Vocabulary

Dawn Lawrie  Dave Binkley
Loyola University Maryland
Baltimore MD
21210-2699, USA
{lawrie, binkley}@cs.loyola.edu

Keywords: source code analysis tools, natural language processing, program comprehension

Abstract

Maintaining modern software requires significant tool support. Effective tools exploit a variety of information and techniques to aid a software maintainer. One area of recent interest in tool development exploits the natural language information found in source code. Such Information Retrieval (IR) based tools compliment traditional static analysis tools and have tackled problems, such as feature location, that otherwise require considerable human effort. To reap the full benefit of IR-based techniques, the language used across all software artifacts (e.g., requirements, design, change requests, tests, and source code) must be consistent. Unfortunately, there is a significant proportion of invented vocabulary in source code. Vocabulary normalization aligns the vocabulary found in the source code with that found in other software artifacts. Most existing work related to normalization has focused on splitting an identifier into its constituent parts. The next step is to expand each part into a (dictionary) word that matches the vocabulary used in other software artifacts.

Building on a successful approach to splitting identifiers, an implementation of an expansion algorithm is presented. Experiments on two systems find that up to 66% of identifiers are correctly expanded, which is within about 20% of the current system’s best-case performance. Not only is this performance comparable to previous techniques, but the result is achieved in the absence of special purpose rules and not limited to restricted syntactic contexts. Results from these experiments also show the impact that varying levels of documentation (including both internal documentation such as the requirements and design, and external, or user-level, documentation) have on the algorithm’s performance.

1 Introduction

Informative identifiers are made up of full (natural language) words and meaningful abbreviations. Readers of programs typically have little trouble understanding the purpose of identifiers composed of full words. In addition, those familiar with the code can (most often) determine the meaning of abbreviations. However, when faced with unfamiliar code, abbreviations often carry little useful information. Furthermore, tools that attempt to exploit the natural language information found in code have a hard time in the presence of abbreviations. One approach to providing meaning to programmers and tools is to translate (expand) abbreviations into full words. Building on work that breaks an identifier into its constituent parts (soft words) [20], this paper presents a methodology for expanding identifiers by expanding the constituent soft words. For example, using vocabulary extracted from the program and its documentation, plen is expanded to prefix length, and using phrase extraction, the acronym cwd is expanded to current working directory.

Identifier expansion is a key part of the USIR project, which aims to improve the productivity of software developers and maintainers through better tool support. The primary objective of the project is to create and improve tools by leveraging techniques from the IR community. Best known for its use by search engines on the Internet, IR encompasses a growing collection of techniques that apply to large repositories of natural language [26]. Recent research has shown that software systems contain significant and useful natural language [2, 8, 22, 23, 28, 33, 34]. Exploiting this information, IR-based tools have tackled problems previously requiring considerable human effort. Examples include (re)establishing links between a program and its documentation [2], developing software metrics [30], and performing feature location [33, 34].

Unfortunately, most tools that leverage IR techniques make the implicit assumption that software artifacts (particularly source code) contain exploitable natural language information. However, at present, there is a debilitating mismatch between the vocabulary used in source code and that used in other software artifacts (e.g., the design document and change requests). The mismatch stems from identifiers being written in what amounts to a different language than
the rest of the documentation, as they include significant abbreviations and acronyms [19]. The negative impact of this mismatch comes from the implicit assumption of most IR techniques that the same words are used whenever a particular concept is described [26].

Previous study has shown a wealth of natural language information within the source code, but also just how different this vocabulary is from that used in other software documents [22]. Thus, there is a need for vocabulary normalization to reap the full benefit of IR-based techniques. Normalization brings the vocabulary of the source code and documentation in line with each other, making it more appropriate for consumption by IR-based tools. Normalization allows tools to better and automatically exploit natural-language semantic information contained in a program.

After presenting background material in Section 2, and before considering related work and future challenges in Sections 6 and 7, this paper makes the following contributions:

1. It refines the vocabulary normalization algorithm, Normalize [20]. This algorithm involves two key tasks: splitting and expansion. The refinements, described in Section 3, support a general approach to effective expansion.
2. Section 4 details the construction of the expansion algorithm that, mirroring the process of statistical machine translation [17], exploits co-occurrence data to select the best of several possible expansions.
3. Finally, Section 5 empirically evaluates the expansion’s prevalence and correctness, finding that, despite being very general in its approach, Normalize correctly expands up to 66% of the identifiers it is applied to.

2 Background

This section first introduces some necessary terminology used to describe identifiers and the two subtasks that make up normalization. When applying IR to software engineering artifacts, the first step is to separate the input text into words (or other atomic units). For natural language documents, such as the requirements or the design, word separation is often straightforward. For example, many languages use separators such as white-space. However, there is no simple means of separating identifiers into individual words, which poses a problem to the adaptation of IR algorithms to the software domain. This problem arises because most IR algorithms make the (often implicit) assumption that the same or similar collection of words is used to describe a concept in all documents.

For source code documents, word breaks (e.g., underscores and camel-casing) are referred to as division markers, while the strings of characters between division markers and the endpoints of an identifier are referred to as hard words. For example, sponge_bob and spongeBob include the hard words sponge and bob. Sometimes splitting into hard words is sufficient (e.g., when all hard words are dictionary words); however, other times hard word splitting is not sufficient, as with identifiers composed of juxtaposed lowercase words (e.g., spongebob). In this case, further division is required. The resulting strings of characters are referred to as soft words. Thus, a soft word is the entire hard word or a sub-string of a hard word. Take, for example, the identifier hashtable_entry. This identifier consists of one division marker (an underscore) and thus, two hard words: hashtable and entry. The hard word hashtable is composed of two soft words, hash and table, while the hard word entry is composed of a single soft word.

Identifying hard words is not always a trivial task. For example, multiple adjacent uppercase letters in camel-cased identifiers can lead to two possible splits – one where all uppercase letters make up a hard word and one where the final uppercase letter is part of the succeeding hard word. In the results reported herein, division markers exist if the identifier includes an underscore, there is a transition from lowercase to uppercase letters, or there is a transition from an alphabetic character to a digit and vice versa. The less obvious cases are left for the more sophisticated soft word splitter [20].

Motivation for splitting beyond the hard word level comes from several past efforts. In applications of IR techniques to source code, splitting is generally based solely on division markers [1, 2, 9, 31]. Such limited splitting has been noted by some authors to be insufficient. For example, Zhao and Zhang note that “It should be indicated that this relatively simple preprocessing [splitting at word markers] is not enough for further use of IR. For instance, identifiers like feature_location and floc need some more sophisticated word recognizers. In our experiment, we preprocess such cases manually · · · ” [34].

The need for such manual preprocessing illustrates both tasks undertaken by normalization. The splitting subtask separates the hard word feature_location into the soft words feature and location and the hard word floc into the soft words f and loc. Here the identifier floc presents a greater challenge. Once correctly split, an IR-based tool would still have difficulties because IR techniques tend not to equate f and feature or loc and location. To establish the correct link, the soft words need to be expanded. Part of the challenge here is to avoid alternate realistic expansions such as file lines-of-code for f loc.

3 Algorithm

This section presents a refinement of the normalization algorithm [20]. As described above, normalization has two tasks: splitting an identifier into soft words and expanding those soft words to associate a meaning (e.g., a dictionary word) with each. While there are advantages to performing
s

denotes the set of distinct expansions
sequence of soft words
size identifies the best expansion over all possible splits. In
Figure 1. Its implementation is described in Section 4.


Normalization


normalize

the maximum coherence model [17], is the basis for
create coherent identifiers. One machine translation tech-
ning abbreviations and acronyms into words and phrases that
fier.


Acronym expansion is more challenging. First,
acronyms need to be identified. Then, a phrase needs to
be matched to the acronym. The source code and docu-
mentation can be mined for phrases that may be expansions
for acronyms. For example, "cms" appears within many of
ghostscript's identifiers. A phrase finder [14] uncovers the
text Color Management System (CMS) in the ghostscript
documentation, which is a likely expansion for the identifi-


The heart of normalization is a similarity metric com-
puted from co-occurrence data. This data is used because
it has proven useful in resolving translation ambiguity. In
other words, Normalize relies on the fact that expanded soft
words should be found co-located in general text. To fur-
ther guide the selection, co-occurrence with context infor-
mation is also considered. For example, the soft word dir,
which can expand to direction or directory. If the local con-
text includes the words north, east, west, and south, then
the higher probability of these words co-occurring with di-
rection, as compared to their co-occurring with directory, is
expected to lead to direction being selected as the correct
expansion. Thus, this information helps to ground the ex-
pansions to a context. It also enables expansion of singleton
soft words (i.e., where the entire identifier is a single soft
word such as num).


The co-occurrences data is a general text data set of
over a trillion words extracted by Google and distributed
by the Linguistic Data Consortium [7]. In the imple-
mentation, the function \( sim(w_1, w_2) \) is the conditional
probability \( p(w_1 | w_2) \), which captures a notion of how of-
ten \( w_1 \) occurs in a similar context to \( w_2 \). Pragmatically
this means the two appear in a five word window in the Google
data set. Function \( sim \) is applied to potential expansions
of an identifier’s soft words to measure the likelihood that a
particular expansion is correct.

Similarities are combined by summing their logs (a pro-
cess that avoids underflow when multiplying small proba-
bilities). The combination, captured by the function Cohe-
sion (see Figure 1 Part 1a) attempts to answer the question,
“do these words tend to occur in a similar context?”

In addition to the expansions of an identifier’s other soft
words, the context of an identifier is used to help guide the
expansion. In the algorithm, the context information is sim-
ply the dictionary words found in close proximity to the
identifier. The current implementation takes “close prox-
imity” to be the identifier’s function. The algorithm favors
expansions that co-occur with words from the context.

As an example, consider the identifier strlen where there
are two possible splits: \( Splits(strlen) = \{ str-len, st-len \} \).
Let the possible expansions (translations) of each soft word
be \( E(str) = \{ steer, string \} \), \( E(len) = \{ lender, length \} \), \( E(st) = \{ stop, string, set \} \), and \( E(rflen) = \{ riflemen \} \). For simplic-
ity, assume initially that the set of context information, \( C \),
is empty. The required similarity scores, computed using
the Google data set, are as follows:
Part 1 For a splitting $s$ of an identifier $id$, the functions $Cohesion'$ and $Cohesion$, defined in Part 1a, are used in Parts 1b and 1c to identify the best expansion for a particular splitting of $s$ into soft words $s_1 s_2 \cdots s_n$. Part 2 then selects the best expansion over all possible splittings.

a. For an expansion $e_{i,j} \in E(s_i)$ of soft word $s_i$, $Cohesion$ is the sum of the similarity for $e_{i,j}$ and the words that might occur around it. These words include the possible expansions of other soft words from $s$ and the words from context $C$. The contribution of a particular soft word, $s_k \in s$, is captured by the function $Cohesion'$. Log's are used avoid underflow when multiplying small probabilities.

The $Cohesion$ of expansion $e_{i,j}$ from splitting $s$ in context $C$ is

$$Cohesion(e_{i,j}, s, C) = \sum_{s_k \in s \neq s_i} Cohesion'(e_{i,j}, s_k) + \sum_{c \in C} \log \text{sim}(e_{i,j}, c)$$

where

$$Cohesion'(e_{i,j}, s_k) = \sum_{e \in E(s_k)} \log \text{sim}(e_{i,j}, e)$$

and

$$\log \text{sim}(x, x) = 0.$$  

b. For soft word $s_i \in s$, $Expand'(s_i)$ is the expansion $e_{i,j} \in E(s_i)$ having the maximal $Cohesion$:

$$Expand'(s_i, C) = e_{i,j} \in E(s_i) \text{ s.t. } \forall e \in E(s_i), Cohesion(e_{i,j}, s, C) \geq Cohesion(e, s, C).$$

c. Finally, $Expand(s, C)$ is an expanded identifier composed of the best expansion for each soft word $s_i \in s$:

$$Expand(s, C) = Expand'(s_1, C)Expand'(s_2, C) \cdots Expand'(s_n, C).$$

Part 2 Scoring function $Score$, defined in Part 2a, is used to pick the best expansion for an identifier $id$ in Part 2b.

a. The $Score$ for splitting $s$ is the average similarities computed over all of pairs of expanded words and each expanded word paired with each context word. An average is used to avoid biasing the results toward excessive splitting. For an (expanded) word $w_i$ in $Expand(s, C)$, $Score'(w_i, s, C)$ captures the contribution to the total score of word $w_i$. (Recall that $\log \text{sim}(x, x)$ is defined to be zero.)

$$Score'(w_i, s, C) = \sum_{w_j \in Expand(s, C)} \log \text{sim}(w_i, w_j) + \sum_{c \in C} \log \text{sim}(w_i, c)$$

$$Score(s, C) = \frac{\sum_{w_i \in Expand(s, C)} Score'(w_i, s, C)}{n(n+c)}$$

where

$c$ is the number of words in context $C$ and $n$ is the number of soft words; thus, $n(n+c)$ counts the number of $\text{sim}$ terms summed.

b. Overall possible splits of an identifier $id$ in context $C$, $Normalize(id, C)$ is the expansion that produces the highest score.

$$Normalize(id, C) = Expand(s) \text{ s.t. } s \in Splits(id) \text{ and } \forall s' \in Splits(id), Score(s, C) \geq Score(s', C)$$

Figure 1. Normalize Procedure
\[
\begin{align*}
\log \ sim(\text{stop, riflemen}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{string, riflemen}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{set, riflemen}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{steer, lender}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{steer, length}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{string, lender}) &= 4.5384 \times 10^{-8} \\
\log \ sim(\text{string, length}) &= 0.0045585
\end{align*}
\]

To begin with, consider the first splitting of \( s \), \( \text{str-len} \). Using the above similarities, \( \text{Cohesion}'(e_{i,j}, s_k) \) where \( e_{i,j} = \text{string} \) and \( s_k = \text{len} \) is computed as follows

\[
\text{Cohesion}'(\text{string, len}) = \log \ sim(\text{string, lender}) + \log \ sim(\text{string, length}).
\]

Next, the computation of \( \text{Cohesion}(\text{string, str-len, } \emptyset) \) sums \( \text{Cohesion}' \) values over all the soft words of \( s \) other than \( \text{str} \) together with string’s similarity with the context words. With no context words and only one other soft word, this simplifies to considering only the remaining soft word, \( \text{len} \),

\[
\text{Cohesion}(\text{string, str-len, } \emptyset) = \text{Cohesion}'(\text{string, len}).
\]

Maximal cohesion is used in Part 1b to choose among the possible expansions. Given

\[
\text{Cohesion}(\text{string, str-len, } \emptyset) = -32.1713
\]

\[
\text{Cohesion}(\text{steer, str-len, } \emptyset) = -48.7865,
\]

\( \text{Expand}'(\text{str}) = \text{string} \) because it has the higher cohesion value of -32.1713. In a similar manner length maximizes cohesion for \( \text{len} \). Thus \( \text{Expand}'(\text{str-len, } \emptyset) = \text{string-length} \). A similar computation is undertaken for the splitting \( \text{st-len} \).

Finally, \( \text{Normalize} \) identifies the split whose expansion has the highest score. Using the equation in Part 2a, \( \text{str-len} \) produces the score \(-7.778064037 \), while for \( \text{str-len} \) the score is \(-24.39324099 \). Therefore, in Part 2b \( \text{Normalize}(\text{str-len}) \) selects \( \text{string-length} \) because it has the highest score.

Hypothetically, if the context \( C \) included the right words (e.g., gun lobby, shooting, target, etc.) then the contribution of the co-occurrences summed over \( c \in C \) could lead \( \text{Normalize} \) to select stop-riflemen instead of \( \text{string-length} \). On the other hand, a context \( C = \{ \text{iterate, over, string} \} \) would reinforce the selection of \( \text{string-length} \).

4 Implementation

The efficient implementation of \( \text{Normalize} \) requires the consideration of three issues: how many splittings to consider, how to determine the set of expansions returned by \( E \) for a given soft word, and the actual computation of the similarity data. To begin with, considering all possible splits is expensive because the set \( \text{Splits}(id) \) includes an exponential number of possible splits. Given that \( \text{GenTest} \) has high recall [20], two sets of possible splits are considered in the experiments. The first includes \( \text{GenTest} \)'s top ten ranked split and the second includes only \( \text{GenTest} \)'s top-ranked split.

The second issue is the mapping from abbreviations to full words. The experiments consider several vocabularies (sets of full words). For each, the key question is how to map the words of the vocabulary to their potential abbreviations. In the experiments, a two phase approach is taken. In the first phase wildcard expansion is applied to an abbreviation to create a list of candidate expansions. Then, the candidate set is filtered by applying one of the following rules from the work of Madain et al. [25] where \( abbr \) is the abbreviation and \( expand \) is a candidate expansion.

1. \( abbr = \text{Truncate}(\text{expand}) \)
   (e.g., \( \text{arg} = \text{Truncate}(\text{argument}) \))

2. If \( \text{Final}\_\text{Character}(abbr) = \text{Final}\_\text{Character}(expand) = s \), then \( \text{Remove}\_\text{Final}\_\text{Character}(abbr) = \text{Truncate}(\text{Remove}\_\text{Final}\_\text{Character}(expand)) \)
   (e.g., \( \text{Remove}\_\text{Final}\_\text{Character}(\text{args}) = \text{Truncate}(\text{Remove}\_\text{Final}\_\text{Character}(\text{arguments})) \))

3. \( abbr = \text{Remove}\_\text{Character}(\text{expand}) \)
   (e.g., \( \text{st} = \text{Remove}\_\text{Character}(\text{set}) \))

4. \( abbr = \text{Remove}\_\text{All}\_\text{Vowels}(\text{expand}) \)
   (e.g., \( \text{mv} = \text{Remove}\_\text{All}\_\text{Vowels}(\text{move}) \))

5. \( abbr = \text{Remove}\_\text{Character}(\text{Remove}\_\text{All}\_\text{Vowels}(\text{expand})) \)
   (e.g., \( \text{ptr} = \text{Remove}\_\text{Character}(\text{Remove}\_\text{All}\_\text{Vowels}(\text{pointer})) \))

If one of the rules produces a match between abbreviation, \( a \), and candidate expansion word, \( w \) then \( w \) is included in \( E(a) \). One exception was introduced to weed out plural forms of words for abbreviations that do not appear to be plural. A second exception forces no expansions for single letter abbreviations. However, the single letter is included in \( E(a) \). This provides a fallback if no suitable expansion can be found.

The final issue is the construction of the similarity data. The function \( sim(x, y) \) is computed from the 5-gram Google data set [7]. This data set contains a myriad of words, many of which cannot possibly be expansions because of the limited characters set used in C-identifiers. The data used in the experiments includes all dictionary words not found on a natural english stoplist. This design decision excluded some pairs later discovered to be of possible interest. In particular the word \( is \) is on the stoplist and occurs in many identifiers. The absence of this word from the similarity information undervalues pairings that include \( is \).
5 Evaluation

This section describes the empirical evaluation of Normalize in three steps. The first describes the subject programs used, the second lays out the design of the empirical study, and finally, results from the study are presented and discussed.

5.1 Subjects

Two programs are considered as case-study test subjects: which version 2.20 and a2ps version 4.14. These programs are from different domains and have different ages. As an older and “low level” program, which makes heavy use of kernel and system calls. As a middle age program, a2ps is a classic command-line application that formats text into PostScript. Table 1 provides some statistics for the two programs including two measures of their size (lines of code (LoC) and non-comment non-blank LoC (SLoC) [1]) and the number of unique identifiers found in each program.

To create an oracle for use in the evaluation, identifiers were expanded by hand. All of which’s 487 identifiers were expanded by one of the two authors. This process included some overlap. Agreement was essentially 100%, obviating the need for replication in the oracle construction. Since a2ps is a larger program, it was not practical to perform exhaustive by-hand expansion. Thus, a random sample of identifiers was chosen and expanded. Table 1 includes the number of identifiers considered in the test oracle as well as the number of hard and soft words these identifiers produced based on the oracle expansions.

5.2 Design

The design of the study addresses four research questions (RQ).

RQ1. What is the overall accuracy of Normalize?

RQ2. Does the vocabulary used have a significant impact on the expansion’s accuracy?

RQ3. Can splitting be improved by better integration with the expander?

RQ4. Can expansion be improved by better integration with the splitter?

Research Question RQ1 considers the accuracy of Normalize at two levels: first the identifier-level where an identifier must be expanded completely correctly and then at the soft-word level where “partial credit” is given for each soft word that is correctly expanded. The second research question considers the impact that the source of the vocabulary has on accuracy, which is also measured at the same two levels. The final two research questions are designed to measure the gap between theoretical best performance and the current implementation.

The identifiers considered break into three groups: standard library calls (those with UNIX man pages), names from standard header files (found by searching /usr/include) together with language keywords, and domain names (the remaining variables were assumed to represent program-specific concepts). Preliminary analysis of variables from the first two groups led to their being removed from further consideration. This set includes keywords (such as void and char), and standard header file values (such as S_ISOCK and uid), conventional names (such as argc and argv), and library routines (such as strcmp). Normalize does well on the variables of these sets for two main reasons. First, many need no splitting or expansion (e.g., main and unsigned), and second, others use simple vocabulary (e.g., longopts expanding to long options). Failures in this group include, for example, strcat and strcmp not being expanded to string-concatenation and string-compare. In such cases, the vocabulary needed for the expansion is not found in the program or its documentation. While this could be fixed through the inclusion of vocabulary extracted from an introductory text book on C programming, the expansion of these identifiers is of little value to non-beginning programmers. To avoid these identifiers from diluting the results, they are not considered further.

Filtering removed 152 of which’s 487 identifiers and 46 of a2ps’s 212 identifiers. It leaves identifiers that would not be easily understood by someone with a general background in C programming, but no domain knowledge. The expansion of these variables should help such a programmer.

5.3 Results

The domain-specific identifiers from the third group are used to address the four research questions. First, for Research Question RQ1, Figure 2 shows the performance of Normalize averaged over the seven vocabulary sets used in the study (the investigation of RQ2 defines the seven vocabularies and considers the impact of each separately). Overall, Normalize performs better on which than on a2ps. This may be because the vocabulary used in the source and documentation are more likely to contain the natural language
words used in expansions for which. By contrast, the vocabulary required to expand abbreviations found in a2ps frequently occurred only in external sources. In addition, though it comes as no surprise, Normalize performs better on both programs when allotted partial credit (accuracy computed at the soft-word level). Finally, expansion performs better when given only GenTest’s top splitting (black bars) rather than the set of the top ten splittings (gray/blue bars). The implications of this observation are considered when discussing RQ4.

The data illustrate that a general expansion algorithm for the automatic expansion of abbreviated identifiers is attainable. In the best case Normalize correctly expands over two-thirds of the soft words for which when working from GenTest’s top split. There is, however, room for improvement. For example, looking at correctly expanded identifiers for a2ps, when Normalize is given less guidance from GenTest only 12% of the identifiers are (completely) correctly expanded. The implications of the range of success levels is considered when investigating Research Questions RQ3 and RQ4.

The second research question considers the impact of the vocabulary on accuracy. This is done by considering combinations of three potential vocabulary sources. The first, denoted with the letter (S), is the source code itself including other variables and comments. The second, denoted with the letter (D), is the internal documentation such as test descriptions, change logs, design documents, etc. Finally, the third, denoted with the letter (M), includes the user manual and other user-level documentation and help pages. In addition to each of these three considered independently, all pairwise combinations (SD, SM, and DM) are used as well as the union of all three (SDM).

The expectation here is that the vocabulary found in the user manual will be the least useful but may include some domain terms. Likewise the vocabulary from the source code will include some useful vocabulary, but also some unwanted words from, for example, programmer observations. Thus, it is expected that the documentation (set D) will contain the best balance of correct words without including too much unwanted vocabulary.

The data used to address Research Question 2 is shown in Figure 3. It includes data for both a2ps and which and also shows the data measuring accuracy by comparing entire identifiers (shown using solid lines) and by comparing the constituent soft words (shown using dashed lines). Finally, the black lines show that the expansion’s performance when using only GenTest’s top splitting rather than letting it work with the top ten ranked splittings, which is shown in blue/gray.

Partial credit (where accuracy includes having some expanded soft words match the oracle) always produces higher accuracy: in Figure 3 the dashed lines are always higher than their solid counter parts. As will be discussed in support of Research Question 4, when the expansion only considers the top-ranked split, it always performs better. Over both programs this improvement is about 15%. In contrast, when expansion uses the top ten splittings, the difference ranges from 20% to 22%, with which having the higher average difference.

Likewise, starting from only GenTest’s top split (the black lines) always produces better results than the corresponding blue/gray line, which represents expansion starting with the top ten splittings. Finally, as was true in the averages shown in Figure 2, the results for which are 20% to 40% higher than those for a2ps.

Combinations of the three choices yields eight comparisons. Each of the resulting eight lines, shown in Figure 3, yields an answer to Research Question RQ2. All four of the
lines for which show the expected pattern where the vocabulary for the documentation is the right balance of desired vocabulary without extraneous words. Interestingly, combinations that include D do not also score well, which suggests that when the vocabulary D is combined with other sources, the other sources provide extraneous vocabulary. Unfortunately, this extraneous vocabulary includes unwanted high probability words.

The data for a2ps is less conclusive. In the four lines shown in the left of Figure 3, the highest accuracy comes from the vocabularies DM, SM, M, and SD. However, the variation is quite small. Furthermore, while D is never the highest, for the two based on GenTest’s top split, it ranks second. This suggests that in the other two cases, extraneous vocabulary is leading Normalize to select (and incorrectly expand) one of the other splits.

Overall there is little support for Research Question RQ2. In particular, each of the eight lines shown in Figure 3 is rather flat. Inspection of the incorrect expansions suggests that Normalize needs to do a better job of excluding general words that have high similarities. One technique that has worked well in the past for identifying words that are descriptive of a particular document is $tf-idf$ [26]. This may help to choose better candidates for expansion. Future work will use such metrics in an attempt to better focus the vocabulary.

Next, Research Question RQ3 asks if splitting can be improved by better integration with the expander. Support would come from GenTest not always choosing the correct split as the basis for expansion. To address this question, each identifier was examined and the correct splitting manually chosen. The resulting data, referred to as the oracle split, is used to evaluate, post expansion, how well GenTest does at choosing the best splitting. Turning this around allows a prediction of how much improvement is possible for GenTest. Figure 4 shows four comparisons. Each pair of bars shows the correctness attained using GenTest’s top ranked splitting and the correctness attained using the oracle split. Note that the oracle split is not automatically attainable. Rather, it provides a best-case scenario. The differences from left to right are 17%, 23%, 17%, and 18%. Thus, overall the answer to Research Question RQ3 is that better integration could improve the splitter, but by no more than about 20%.

Finally, turning to Research Question RQ4, expansion can be improved by better integration with the splitter if it can exploit the metrics used by GenTest in the expansion process. That there is room for improvement is seen in all cases where the expansion does not select GenTest’s top-ranked split and then goes on to produce the wrong answer. In these cases the expander is going astray because it has naively disregarded some information exploited by GenTest. This final research question is answered in the affirmative: when Normalize does not follow GenTest’s advice and selects a splitting other than the top-ranked splitting, it goes on to produce the wrong answer 63% of the time. It does this even though it produces the correct expansion when it started with the top-ranked split.

Given that the experiment presented is a proof of concept, it is premature to consider threats to validity in depth. Some of the more evident threats include the limited number of programs considered and their both being C programs. In addition the oracle data set was manually produced by the authors. Finally, the current implementation takes about eight seconds to generate an expansion for an identifier.

6 Related Work

This section briefly considers the broader category of IR-based tools giving an example before considering the three existing previously published expansion algorithms. Many of the challenges faced in software engineering can be (partially) addressed using IR techniques applied to source code and its associated documents. Example applications include traceability link recovery [2], concept or feature location [34, 33], reverse engineering [15], impact analysis [8], software clustering metrics [28], software libraries [30], developer identification [23], and software comprehension [9]. Of these application categories, over half are strongly dependent on the language contained in identifiers.

An example application comes from metrics used to assess design quality, predict software quality, identify fault prone modules, and identify reusable components. Existing (non-IR) metrics are primarily based on structural aspects of software, such as the number of attributes in a class, the number of lines of code, etc. Recently, Marcus et al. defined a coupling metric between classes in terms of conceptual similarity between each classes’ methods [28]. For two methods, cosine similarity between the natural language found in the two is used to establish the similarity. However, they found that when two conceptually related classes
used even slightly different abbreviations for concepts, the cosine similarity under-represented the true coupling. The vocabulary normalization techniques presented herein will help such an approach by replacing abbreviations with their normalized equivalent. This will generate a more accurate approximation of the true coupling.

The need to normalize vocabulary in support of IR-based tools was first noted by Feild et al. [13, 21]. This early work focused on the need to split hard words into multiple soft words. It did incorporate a limited form of wildcard expansion: when there was a single match, it was returned as the expansion for the soft word. This algorithm worked independently on each soft word and simply failed when zero or more than one match occurred. In the later case, this was because there was no way to select between the possible expansions. Given these restrictions the approach still managed to correctly expand 40% of a sample of 64 identifiers. In comparison, Normalize uses co-occurrence information about possible expansions and the context to help make this choice.

Since this initial work, two improvements have been investigated. The first works with Java code, where it applies a specific series of regular expression searches in an ever expanding syntactic context [16]. This starts with the JavaDoc comments where, for example, the pattern

```java
@param abbreviation abbreviation[a-z0-9A-Z]*
```

is used to expand an abbreviation formed by truncating the expanded word. For example, this search succeeds in expanding the abbreviation `len` when the JavaDoc includes the comment `@param len - length of the wall`. This approach works well, correctly expanding 60% of 250 non-dictionary soft words extracted from Java identifiers. Increasing the correctness would be possible if the vocabulary needed for an expansion could be selectively acquired. For example, such vocabulary is often found in a class or file defining a type rather than at the type’s point of use, or, looking even further a field, in the program’s documentation. The challenge with incorporating wider sources of information is filtering out irrelevant vocabulary. Normalize exploits co-occurrence information to filter out such words.

The second alternate approach applies dynamic time warping to split and expand identifiers [25]. Dynamic time warping aligns two signals (classically two speech utterances) by “warping” the time when certain attributes of the speech occur. Applied to an abbreviation and a potential expansion the warping is used to align the letters of the abbreviation with those of a potential expansion. The technique requires a reasonably precise dictionary because an abbreviation such as `len` is easier to warp into `letter` than length. It is possible that the co-occurrence data used by Normalize could be used to guide the warping or even pre-process the dictionary to limit it to words co-occurring with words found in a program.

7 Future Challenges

Future work, already underway, will consider improvements to the vocabulary normalization algorithm’s two phases and its impact on IR tools. An example improvement is expected from the incorporation of relative entropy [10, 27]. Such a probabilistic approach is well suited for automatic abbreviation and acronym detection. For example, n-gram language models [29], built from n sequential letters, have been successfully used in speech recognition software [27]. In essence, relative entropy identifies unusually frequent sequences of characters. These are likely to be meaningful in a particular document. For example, “ems” appears within many of ghostscript’s identifiers. Since it has a high relative entropy, it should be identified as an acronym used in the program. Adding a relative entropy metric should benefit GenTest. Furthermore, combined with a phrase finder [14], it should allow the automatic expansion of acronyms.

With the normalization implementation in place, empirical investigation of its impact on existing tools will begin. Tools and techniques to be experimented with will be drawn from the collection of problems in software engineering to which IR has been applied [4, 5, 12]. Vocabulary normalization is expected to dramatically improve existing (and future) IR-based tools. As an illustration of the impact that Normalize can have, consider the FLAT-3 [32] feature locator. In addition to dynamic tracing, this tool indexes the current Eclipse workspace and then uses cosine similarity to determine how close a query is to any class, method, or other file in the workspace. A preliminary experiment considered searching for the concept “account number”, and produced the following improvement in FLAT-3’s performance. Before normalization, this search returns the correct file that contains `accountNum` with a confidence (cosine similarity) of 0.60. After normalization, which replaces `accountNum` with `accountNumber`, the confidence jumps to over 0.95. Similar improvement is expected from other IR-based SE tools.

8 Summary

This paper describes the first general purpose expander. Unlike its predecessors which had rather strict requirements on the natural language source (e.g., the need for JavaDoc comments or a special purpose dictionary), Normalize can expand abbreviations from arbitrary source code. In addition, although internal and external documentation can be good sources for vocabulary, the performance of Normalize does not degrade significantly when only the source code is available as a vocabulary.

Looking forward, vocabulary normalization is a preprocessing step that allows source code to satisfy the often implicit assumption of IR-based techniques that the same words are used whenever describing a particular concept.
This assumption is commonly violated by the language used in source-code identifiers when compared to that used in other software documents. The vocabulary normalization described in this paper is a key step to improving existing and future IR-based tools and techniques. Normalization has two aspects: splitting and meaning assignment. It proves correct up to 66% of the time. Although this may produce data too noisy for human consumption, IR techniques have recently been shown to be robust in the face of noisy speech recognition, which means that the noise may not hamper improved tool performance [1]. Further evolution of the algorithms for both phases and continued empirical study are expected to show continued improvement in precision and the benefit of vocabulary normalization.

9 Acknowledgments

Thanks to Alex Campbell, David Christo, and Christopher Uehlinger for their work on the exposition. Thanks to the anonymous referees for their suggestions, which contributed to substantial improvement in the paper. Support for this work was provided by NSF grant CCF 0916081.

References


