TR120625-42: Vocabulary Normalization’s Impact on IR-Based Concept Location

Christopher Uehlinger  
Computer Science Department  
Loyola University Maryland  
Baltimore, USA  
cuehling@cs.loyola.edu  

Dave Binkley  
Computer Science Department  
Loyola University Maryland  
Baltimore, USA  
binkley@cs.loyola.edu  

Dawn Lawrie  
Computer Science Department  
Loyola University Maryland  
Baltimore, USA  
lawrie@cs.loyola.edu

Abstract—Tool support is crucial to modern software development, evolution, and maintenance. Early tools reused the static analysis performed by the compiler. These were followed by dynamic analysis tools and more recently tools that exploit natural language. This later class has the advantage that it can incorporate not only the code, but artifacts from all phases of software construction and its subsequent evolution.

Unfortunately, the natural language found in source code often uses a vocabulary different from that used in other software artifacts and thus raises the vocabulary mismatch problem. This problem exists because many natural-language tools imported from Information Retrieval (IR) and Natural Language Processing (NLP) implicitly assume the use of a single natural language vocabulary. Vocabulary normalization, which goes well beyond simple identifier splitting, brings the vocabulary of the source into line with other artifacts. Consequently, it is expected to improve the performance of existing and future IR and NLP based tools. As a case study, an experiment with an LSI-based feature locator is replicated. Normalization universally improves performance. For the most common IR-based approaches, this improvement is over 200% ($p < 0.0001$).

Keywords—vocabulary normalization; information retrieval; concept location

I. INTRODUCTION

Modern software evolution 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) and Natural Language Processing (NLP) based 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 [2], [27], assessing program quality [30], and performing concept location [34], [37]. This paper takes as a representative example a tool that performs one of the most common software maintenance activities: concept location [10]. Example concept-location tools include FLAT [33], Google Eclipse Search [32], and JRipples [5].

Best known for its use by search engines on the Internet, IR encompasses a growing collection of statistical techniques that apply to large repositories of natural language text [26]. In contrast, NLP techniques attempt to uncover a more semantic understanding such as part-of-speech information. Recent research has shown that software systems contain significant, useful natural language [2], [7], [22], [23], [28], [34], [37].

To reap the full benefit of IR-based and NLP-based techniques, the language used across all software artifacts (e.g., requirement and design documents, test plans, as well as the source code) must be the same. Unfortunately, there is a significant proportion of invented vocabulary in source code in the form of abbreviations and acronyms. In effect, source code uses a different language than other software artifacts [17]. The negative impact of this vocabulary mismatch stems from the implicit assumption of IR and NLP techniques that the same words are used whenever a particular concept is described [26]. NLP techniques go a step further in assuming that these words are from a natural language. Vocabulary normalization brings the vocabulary of the source code in line with that of a project’s other artifacts, making it more appropriate for use with IR-based tools (in the remainder of the paper references to IR implicitly include NLP-based tools as well).

The goal of the NORM project is to develop techniques for normalizing software vocabulary to both increase ease of programmer comprehension and to improve tool performance. Doing so will improve the productivity of software engineers. While a simplification, normalization can be seen as a two-step process. The first step, identifier splitting, has seen significant attention [12], [11], [20], [14], [6]. However, prior empirical study has indicated that splitting alone can be insufficient [9] (a result confirmed in Section V). Normalization’s second step, expansion [18], [16], [20], has received less attention. In particular, it has received insufficient empirical study as to its impact on IR-based tools. Normalization (splitting and expansion) has shown promise [19] and thus deserves further study.

After presenting the Normalize algorithm in Section II and before considering related work in Section VI, the paper’s three core contributions appear in Sections III, IV, and V. These sections describe the three steps of the replication of an earlier empirical study that applied Latent Semantic Indexing (LSI) to the problem of locating concepts in NCSA Mosaic.
II. Normalize

When retrieving documents using a search engine, having an accurate estimate of the importance of a query word is paramount to achieving good performance. In particular, if the importance of a query word is underestimated for a particular document, that document can be buried among the non-relevant documents. When applying IR techniques to source code, this underestimation is a great concern because programmers habitually use personal abbreviations and acronyms. Normalize seeks to remedy this situation by breaking identifiers up into parts and then expanding any abbreviations and acronyms to full words. After replacing identifiers with their normalized versions, IR techniques will be able to better estimate each document’s important vocabulary.

This section briefly presents the normalization algorithm [19]. It is first necessary to define some terminology. Within an identifier, 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 terms. For example, the identifiers sponge_bob and spongeBob include the hard terms sponge and bob. Sometimes splitting into hard terms is sufficient (e.g., when all hard terms are dictionary words); however, other times hard term splitting alone is insufficient, 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 terms. Thus, a soft term is either the entire hard term or a sub-string of a hard term. Take, for example, the identifier hashtable_entry. This identifier consists of one division marker (an underscore) and thus, two hard terms: hashtable and entry. The hard term hashtable is composed of two soft terms, hash and table, while the hard term entry is composed of the single soft term, entry.

Vocabulary normalization involves two tasks: splitting an identifier into soft terms and expanding those soft terms to associate a meaning (e.g., a dictionary word) with each. Task 1 is to separate hard terms into soft terms composed of character sequences that represent words, abbreviations, or acronyms. Of several existing splitters [12], [11], [20], [14], [6], GenTest [20] is used to accomplish Task 1. While empirical evidence finds each of the splitters having its niche, overall GenTest and Samurai [11] are the top two performers [15].

GenTest uses the generate and test style. The generation part of algorithm is simple as it generates all possible splittings. While this generates an exponential number of splittings, most identifiers are short and thus the computational effort is not excessive. The test part is more complex, although it is quite efficient, as it simply evaluates a scoring function against each proposed splitting. The complexity is within this function, which is a linear combination of metrics that describes the quality of the split. The construction of this function is described by Lawrie et al. [20]. This scoring function provides a ranking for each possible split of an identifier, which is utilized in Task 2.

Task 2 assigns a meaning to each soft term. There are two cases: the first is the easy case where the soft term is a dictionary word and thus has a well-established meaning. In the second case, non-dictionary words are assumed to be abbreviations or acronyms. A naive, but illustrative algorithm for expanding an abbreviation is based on finding a list of candidate expansions and then applying ideas taken from machine translation. For example, given the abbreviation horiz, the code and the external documentation are searched for strings matching “h*r*i*z””, where a “*” represents an arbitrary sequence of letters, to establish candidate expansions. A search of the program from which horiz was extracted uncovers a unique match: the dictionary word horizontal. Thus, the meaning of the abbreviation horiz is equated to that of the word horizontal.

Acronym expansion is more challenging. First, acronyms need to be identified. Then, a phrase needs to be matched to each acronym. The source code and documentation can be mined for phrases that may be expansions for acronyms. For example, “cms” appears within many of the program ghostscript’s identifiers. A phrase finder [13] uncovers the text Color Management System (CMS) in the ghostscript documentation, which is a likely expansion for the the soft term “cms”.

Machine translation techniques offer a means of expanding abbreviations and acronyms into words and phrases that create coherent identifiers. One machine translation technique, the maximum coherence model [31], is the basis for the Normalize algorithm.

Normalize identifies the best expansion over all possible splits. From a set of candidate splits Splits(id) of identifier id, a split s ∈ Splits(id) is composed of a sequence of soft terms s1s2 · · · sn, where S denotes the set of soft terms. For soft term si, E(s), denotes the set of distinct expansions c1s1, c2s2, · · · , cmsm for si.

The heart of normalization is a similarity metric computed from co-occurrence data. This data is used because it has proven useful in resolving translation ambiguity [31]. In other words, Normalize relies on the fact that expanded soft terms should be found co-located in general text. To further guide the selection, co-occurrence with context information is also considered. For example, the soft term dir may expand to direction or directory. If the local context includes the words forward and backward, a higher probability of these words co-occurring with direction, 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 expansions to a context. It also enables expansion of singleton soft terms (i.e., where the entire identifier is a single soft term such as num). In the
algorithm, the set of context words, denoted $C$, is simply the dictionary words found in close proximity to the identifier. The current implementation takes “close proximity” to be the identifier’s function.

The co-occurrence data is a general text data set of over a trillion words extracted by Google and distributed by the Linguistic Data Consortium [4]. Normalize uses the co-occurrence probability, denoted $p(w_1 \cap w_2)$, which is computed as the number of times $w_1$ and $w_2$ occur together in a five word window in the Google data set divided by the total number of word-pairs in the Google data set.

The fitness of a particular expansion, $e_k$, for $s_k$, is determined by summing the logs of the probabilities of the expansion occurring with all the expansions, $E(s_i)$, of the other soft terms, $s_i \in S - \{s_k\}$ and the context words, $C$. This fitness measure attempts to answer the question, “do these words tend to occur in a similar context?”

Given a particular split of an identifier, $s \in Splits(id)$, Normalize replaces each $s_i \in s$ with $E_i$, the expansion, $e_i$, that maximizes the fitness measure; thus, the expansion of $s$ is composed of the sequence of expansions $E_1E_2 \cdots E_n$. Finally, the best expansion is chosen from among the candidate splits. The fitness used in this selection is also a sum of log probabilities. In this case the sum is over all pairs from the chosen expansions, $E_i$, and the context $C$.

The efficient implementation of Normalize requires the consideration of three issues: how many splittings to consider, how to determine the set of expansions returned as $E$ for a given soft term, and the actual computation of the similarity data. To begin with, considering all possible splits is expensive because the set $Splits(id)$ includes an exponential number of possible splits. Given that GenTest has high recall [20], Normalize uses GenTest’s top ten ranked splits. It takes about a third of a second to normalize an identifier. This time includes both the splitting process and the expansion process.

The second issue is the mapping from abbreviations to full words. In the first phase wildcard expansion is applied to an abbreviation to create a list of candidate expansions. Then, the candidate set is filtered to remove improbable expansions using the identifier abbreviation rules enumerated by Lawrie et al. [19]. These rules attempt to capture the techniques used by engineers when inventing abbreviations.

The final issue is the construction of the similarity data for computing co-occurrence probabilities from the 5-gram Google data set [4]. This data set contains a myriad of words, many of which cannot possibly be expansions because of the limited character set used in identifiers. Thus, words containing characters other than alphabetic letters are eliminated before word-pairs are counted.

III. STUDY DESIGN

To avoid experimenter bias it is important to replicate an existing study as closely as possible. Doing so avoids, for example, the possibility that in a newly-devised study, queries that favor normalized source code would be chosen. The particular experiment chosen applies Latent Semantic Indexing (LSI) to the problem of concept location [29]. Hereafter this study, performed by Marcus et al., is referred to as the original study.

Concept location, the activity of identifying the location in the source code of a desired functionality, is a frequent and manually time-consuming task for software engineers [10]. Some authors distinguish between concept location and feature location, where the later is essentially, restricted to concepts found in the problem domain [24].

The goal of the study is to investigate the impact of vocabulary normalization (Research Question 1). The study also considers the impact of simply splitting identifiers (Research Question 2). As a preliminary step it is necessary to replicate the conditions of the original study as closely as possible. The replication provides a baseline for measuring the impact of normalization. This section lays out the necessary pieces for replication including the preparation of the corpus, determining the relevant documents, forming the query set, and determining the performance metrics used.

A. Preparation of the corpus

Classically, IR attempts to retrieve documents relevant to a query from a (large) collection of documents. When applying such an approach to concept location, the source code is first broken into chunks that form the IR system’s documents. These documents are then indexed creating a corpus.

In the original study, the source code for NCSA Mosaic was used. This source was first divided into blocks of code. Each declaration, function, and header file formed a separate block. Due to a limitation in the original LSI implementation, functions over 10,000 characters long were split into multiple blocks. The dividing up of the source into blocks was well documented in the original study in a file that contained the name of each source file and the starting line number of each block from within that file. Using this file the same 2347 blocks as used in the original study were extracted.

The second step extracted words (strings of alphabetic characters) from each block. Then, to prevent C keywords from warping the LSI term weights, the extracted words were filtered by removing words found on a C-language stop-list. This left only identifiers and words from comments.

The third step addresses the vocabulary mismatch problem with varying degrees of intervention. The vocabulary mismatch problem arises because, unlike the source code, the queries are written in natural language. The spectrum of solutions considered ranges from conservative splitting where identifiers are split into hard terms at underscores, digits, and transitions from lowercase to uppercase [11], through full normalization where identifiers are aggressively
split and expanded by Normalize. In between, the impact of the representative aggressive splitter GenTest [20] is also considered. Recent empirical evidence finds that existing aggressive splitters provide essentially the same splittings [15].

B. Relevant Documents

The original study considered five documents to be relevant documents. These were identified through manual inspection of the Mosaic source code by the original study’s authors. They used this set of documents to evaluate the accuracy of their approach. However, when comparing their approach with a dependence graph concept locator, it became apparent that two additional documents should have been considered relevant: “The first two [documents] are obviously related [to the concept]” [29].

In Section IV, results are reported for both relevant sets. The set of five is used to more faithfully replicate the conditions of the original study, while the set of seven is used because it is a more appropriate target.

C. Query Sets

The original study made use of two sets of queries: user generated and automatically generated queries. For the user generated queries, Marcus, et al. asked six members of their research lab to formulate a query that describes their research lab to formulate a query that describes generated and automatically generated queries. For the user

Table I

<table>
<thead>
<tr>
<th># Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 font</td>
</tr>
<tr>
<td>2 font size style small regular large</td>
</tr>
<tr>
<td>3 font style large small regular family</td>
</tr>
<tr>
<td>4 font style bold italics large small regular</td>
</tr>
<tr>
<td>5 font size style small regular large family bold italics type</td>
</tr>
<tr>
<td>6 font size style small regular large family bold italics</td>
</tr>
<tr>
<td>7 font family style bold italics size small regular medium large</td>
</tr>
<tr>
<td>8 font size style small regular large family bold italics medium type</td>
</tr>
</tbody>
</table>

provide more extensive queries than is typical. Thus two sets of shorter queries are developed using the ten unique words other than “font” found in the user queries. The first, paired with each of the ten unique words from the original set. The second, triples includes 45 queries composed of the word “font” and two of the ten unique words as are shown in Table II.

D. Performance Metrics

The original evaluation of a tool’s performance was indirectly based on the IR metrics precision and recall. Given a set of relevant documents as well as a set of retrieved documents, precision can be calculated as the percentage of the retrieved documents that are in the set of relevant documents. Recall can be calculated as the percentage of the relevant documents that appear in the set of retrieved documents.

In addition to the two query sets from the original study, two additional sets are used in this study. The motivation for this addition is that the first two query sets are dominated by unnaturally long queries. Furthermore this original set was gathered in the presence of the experimenter, which probably influenced participants behavior leading them to

Table II

<table>
<thead>
<tr>
<th>Pair Queries</th>
<th>Triple Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td># Query</td>
<td># Query</td>
</tr>
<tr>
<td>1 font size</td>
<td>1 font size</td>
</tr>
<tr>
<td>2 font style</td>
<td>2 font size</td>
</tr>
<tr>
<td>3 font small</td>
<td>3 font size</td>
</tr>
<tr>
<td>4 font regular</td>
<td>4 font size</td>
</tr>
<tr>
<td>5 font large</td>
<td>5 font size</td>
</tr>
<tr>
<td>6 font family</td>
<td>6 font size</td>
</tr>
<tr>
<td>7 font bold</td>
<td>7 font size</td>
</tr>
<tr>
<td>8 font italics</td>
<td>8 font size</td>
</tr>
<tr>
<td>9 font medium</td>
<td>...</td>
</tr>
<tr>
<td>10 font type</td>
<td>45 font medium</td>
</tr>
</tbody>
</table>
While this ad-hoc process is replicated for completeness, the more conventional measures for ranked retrieval **Average Precision** (AP) and **Mean Average Precision** (MAP) are also used to measure results [26]. An advantage of using AP is that it does not require the selection of an arbitrary set of retrieved documents. In addition AP and MAP are more appropriate for ranked retrieval because precision and recall do not take the ranks into account. Consider, for instance, the top ten documents in a ranked list, precision and recall remain the same whether the relevant documents are the first five or the last five, even though the former is preferable.

Average precision 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.

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

Calculating average precision involves iterating down the rank list, stopping at each relevant document, and calculating the precision at $k$. These precisions are then averaged.

$$\text{AP} = \frac{\sum_{r \in \text{Relevant}} \mathcal{P}(\text{rank of } r)}{\text{Relevant}}$$

As its name implies, given a set of queries, MAP is the mean of the AP values taken over the set of queries. MAP is the standard metric to report for the performance of an information retrieval system [26]. Given that performance on individual queries is expected to be highly variable [26], MAP better represents an IR system’s overall performance on individual queries is expected to be highly variable [26], even though the former is preferable.

MAP is that it does not require the selection of an arbitrary set of retrieved documents. In addition AP and MAP are more appropriate for ranked retrieval because precision and recall do not take the ranks into account. Consider, for instance, the top ten documents in a ranked list, precision and recall remain the same whether the relevant documents are the first five or the last five, even though the former is preferable.

Average precision 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.

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

Calculating average precision involves iterating down the rank list, stopping at each relevant document, and calculating the precision at $k$. These precisions are then averaged.

$$\text{AP} = \frac{\sum_{r \in \text{Relevant}} \mathcal{P}(\text{rank of } r)}{\text{Relevant}}$$

As its name implies, given a set of queries, MAP is the mean of the AP values taken over the set of queries. MAP is the standard metric to report for the performance of an information retrieval system [26]. Given that performance on individual queries is expected to be highly variable [26], MAP better represents an IR system’s overall performance on individual queries is expected to be highly variable [26], even though the former is preferable.

$$\text{MAP} = \frac{\sum_{q \in \text{Queries}} \text{AP for query } q}{\text{Queries}}$$

Finally, following the advice of Smucker et al. [35] who studied statistical testing of IR results, Student’s $t$-test is used in the analysis. In particular, Smucker et al. found the non-parametric Wilcoxon signed-rank test undesirable and noted that in practice the $t$-test provided a good approximation to the computationally expensive randomization test (permutation test) [35]. To double check the findings presented in Section V the Wilcoxon test was also applied. In all cases the same results were obtained thus confirming the $t$-test findings. Therefore, only the (paired) $t$-tests are reported.

**IV. STUDY REPLICATION**

This section first describes the configuration settings used to replicate the original study. It then considers two minor changes to these settings made to produce a more suitable baseline for examining the impact of vocabulary normalization.

The original study used the 1990 Bell Communications Research implementation of LSI to index the document collection and process the queries [8]. In preparing the corpus (following the steps described in Section III-A), the replication extracted the same blocks of code from the source. From each block, comments and identifiers were extracted. The same thirty-five word stop-list of C keywords was used to filter the extracted words.

After this identifiers in the set of extracted words were split. Unfortunately, insufficient detail about the splitting process was retained with the original study. This means that an identical replication is not possible. However, since this original study, the nuisances of splitting have become better understood [11]. Thus based on the original description, conservative splitting (Section III-A) was used. Finally, the resulting documents (collections of split identifiers) were indexed using the following standard parameter settings “-wlog entropy -kd 0 -kg 0 -P svd.”

Because an identical replication was not possible, empirical data was collected to demonstrate that the differences were minimal. Table III compares the originally published data and that obtained using conservative splitting. As the table shows, these results are very close. Furthermore, while a 100% replication would have been the preferred starting point, for the study of Normalize’s impact, it is actually more important to make use of external relevance judgments. In particular those used in the original study. This is because their use avoids the threat of experimenter bias in the selection of the queries and the relevant documents. Thus, the use of the queries and relevance judgments from the original study assures the unbiased assessment of Normalize’s impact.

<table>
<thead>
<tr>
<th>Investigated Q documents</th>
<th>Recall</th>
<th>Precision</th>
<th>Last relevant document position</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Study – Original Tool Chain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0%</td>
<td>0.00%</td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>100%</td>
<td>9.25%</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>60%</td>
<td>33.33%</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>100%</td>
<td>33.33%</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>79</td>
<td>100%</td>
<td>6.32%</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>57</td>
<td>100%</td>
<td>8.77%</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>100%</td>
<td>10.02%</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>100%</td>
<td>6.94%</td>
<td>18</td>
</tr>
</tbody>
</table>

| Original Study – Current Tool Chain |
|--------------------------|--------|-----------|-------------------------------|------|
| Investigated Q documents | Recall | Precision | Last relevant document position | Step |
| 1 | 2 | 0% | 0.00% | 93 | 1 |
| 2 | 69 | 100% | 7.14% | 26 | 5 |
| 3 | 17 | 80% | 22.22% | 21 | 3 |
| 4 | 24 | 100% | 20.00% | 9 | 4 |
| 5 | 21 | 100% | 22.73% | 10 | 3 |
| 6 | 28 | 100% | 17.24% | 9 | 4 |
| 7 | 18 | 100% | 26.32% | 7 | 4 |
| 8 | 12 | 100% | 38.46% | 7 | 3 |
The first of the two minor changes increases the LSI tool’s maximum document size from 10,000 to 50,000 characters. While this has no effect on the original study because all its documents were forced to be less than 10,000 characters, this extra space is needed for the normalized source code where abbreviation expansion increases the space taken by many of the identifiers.

The second minor change omits the original identifiers from the documents. These identifiers were retained in the original study to improve results for queries that used identifier names from the code. However, when indexed by the concept locator, these additional “words” essentially act as noise that masks the importance of individual words. Fortunately, vocabulary normalization obviates the need to retain the original identifiers with the added benefit that the associated noise is removed. As an example, the second minor change replaces “font_size” with “font-size” rather than “font_size/font-size” and thus omits the original identifier.

The two minor changes have only a slight impact on the tool’s output. This comparison is presented in terms of Mean Average Precision (MAP). This change is made because the format used in the original study (seen in Table III) solves the problem of measuring the quality of a ranked retrieval in an ad-hoc fashion. The use of MAP to measure the performance of IR-based tools is more conventional and thus preferred.

The impact of the two minor changes is shown in Figure 1. From this data, increasing the maximum document size to 50,000 characters has no effect. Furthermore, omitting the original identifiers produces a slight improvement in MAP. This improvement is evidence that the original identifiers were producing unwanted noise.

The configuration “without Original Identifiers,” which also included the 50,000 byte buffer, forms the baseline for the investigation into Normalize’s impact. The data used to compute these MAP values is presented in Table IV. In addition to MAP, the data includes the average precision for each query and the relevant documents and their ranks to support future replication of this work.

V. IMPACT OF Normalize

To evaluate the impact of normalize, the four different query sets described in Section III-C are used with three variants of the Mosaic source. The variants capture a spectrum based on how identifiers are treated. At one end of this spectrum, conservative splitting is used to provide the baseline or original variant. At the other end, Normalize provides the most dramatic treatment. In between the aggressive splitter, GenTest, which has similar or superior precision when compared to other aggressive splitters [15], is used. Comparison’s with GenTest help discern the relative impact of splitting versus expansion. Normalize’s impact is measured relative to the first two variants. This section ends by considering threats to the validity of the replication.

Figure 2 shows the results for all three variants using all four query sets. Considering first the original query set, Table V shows the impact of normalization on the individual relevant-document ranks and the resulting average precisions using both five and seven relevant documents. The resulting MAP values are shown graphically in Figure 2.

In the statistical analysis of the data Normalize’s output is used in two comparisons: with the output obtained using conservative splitting and with the output obtained using GenTest (note that the comparison between conservative splitting and GenTest does not address either of the two research questions and thus is never made). While one could argue that these two comparisons are independent, the analysis takes the conservative approach and applies Bonferroni’s correction for multiple comparisons. Uncorrected the following p-values would be halved.

The improvements for the Original Set seen in Figure 2 are 14 and 10 percentage points for 5 and 7 relevant documents. While this improvement might be considered practically significant, Student’s t-test (paired), applied to the average precisions, shows no statistical difference between conservative splitting and Normalize (p = 0.184) for 5 relevant documents and no statistical difference (p = 0.392) for 7 relevant documents. (As a notational convinence, subsequent p-values for the two sets of relevant documents are reported as “(p = ··· for 5 and p = ··· for 7).”)

Similarly, there is no a statistically significant difference when comparing GenTest with Normalize (p = 0.132 for 5 and p = 0.137 for 7). The p values are slightly lower primarily because the average precisions for GenTest have a lower standard deviation. The lack of statistical significance is perhaps not surprising given the low power available with only eight queries (this first set is the smallest of the four).
Table IV

<table>
<thead>
<tr>
<th>Q</th>
<th>Precision</th>
<th>Relevant Documents (Rank:Document ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>74.55%</td>
<td>1:1468 2:257 3:389 8:390 22:1229</td>
</tr>
<tr>
<td>3</td>
<td>87.14%</td>
<td>1:257 2:1468 3:389 4:390 14:1229</td>
</tr>
<tr>
<td>4</td>
<td>78.93%</td>
<td>1:1468 2:390 4:257 7:389 8:1229</td>
</tr>
<tr>
<td>5</td>
<td>72.95%</td>
<td>1:390 3:1468 5:257 6:389 7:1229</td>
</tr>
<tr>
<td>6</td>
<td>79.62%</td>
<td>1:1468 2:390 5:257 6:389 7:1229</td>
</tr>
<tr>
<td>7</td>
<td>69.62%</td>
<td>1:390 4:1468 5:1229 6:257 7:389</td>
</tr>
<tr>
<td>8</td>
<td>69.62%</td>
<td>1:390 4:1468 5:1229 6:257 7:389</td>
</tr>
</tbody>
</table>

MAP: 68.23% for 5 Relevant Documents

<table>
<thead>
<tr>
<th>Q</th>
<th>Precision</th>
<th>Relevant Documents (Rank:Document ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>90.82%</td>
<td>1:257 2:1468 3:389 4:390 5:391 7:389 14:1229</td>
</tr>
<tr>
<td>5</td>
<td>100.00%</td>
<td>1:390 2:390 3:391 4:257 5:392 5:257 6:389 7:1229</td>
</tr>
<tr>
<td>6</td>
<td>100.00%</td>
<td>1:390 2:390 3:391 4:257 5:392 5:257 6:389 7:1229</td>
</tr>
<tr>
<td>7</td>
<td>100.00%</td>
<td>1:390 2:390 3:391 4:257 5:392 5:257 6:389 7:1229</td>
</tr>
<tr>
<td>8</td>
<td>100.00%</td>
<td>1:390 2:390 3:391 4:257 5:392 5:257 6:389 7:1229</td>
</tr>
</tbody>
</table>

MAP: 83.91% for 7 Relevant Documents

Next turning to the auto-generated query set, also shown in Figure 2, Normalize dramatically improves the performance of the concept locator when processing these queries. For this set, paired t-tests show that Normalize achieves significantly better performance than conservative splitting ($p < 0.0001$ for 5 and $p < 0.0001$ for 7) and GenTest ($p < 0.0001$ for 5 and $p < 0.0001$ for 7). One caveat with this result, while the four query sets generally meet the sample independence assumption of the t-test, this query set is constructed by adding one new word to the end of each query set, which likely impacts the independence assumption. On the other hand t-tests have been shown equivalent to more general tests when applied to IR tools comparisons and thus have been recommended for such comparisons [35].

MAP values for the final two sets, the pairs and triples, are also shown in Figure 2. Given that pair and triple queries are on average significantly shorter in length than queries in the original set, this data demonstrates an interesting trend regarding the performance of the concept locator on
A. Threats to Validity

As a case study, this work has the common standard threats to validity found in similar work. For example, the results may not generalize to other software projects (external validity). Threats particular to this experiment include the atypical use of evaluating the output of all queries against the same sets of five and seven relevant documents. However, threats related to the queries (e.g., their consistency with the source code vocabulary) are more applicable to the original experiment. These are less of an issues here because the same queries are used with all three treatments. Finally, there is little threat of algorithmic bias as the conservative splitting algorithm is well defined [11] and GenTest’s performance is comparable to splitters built by others such as Samurai [15].

VI. RELATED WORK

Vocabulary normalization both splits an identifier into its constituent parts and then expands each part into a full dictionary word to match the vocabulary found in other software artifacts. Most existing work related to Normalize has focused on the first task: splitting an identifier into its constituent parts. However, such work is not the focus of this section as the expansion part of normalization has a much greater impact on the task of concept location.

Because concept location is one of the most common activities performed during software maintenance and evolution, it is a reasonable task on which to consider the impact of vocabulary normalization. This section first briefly describes the range of solutions to the feature-location problem before considering two recent investigations into the impact of human modifications to identifiers on feature location. This is followed by a review of other expansion techniques, and finally recent work on empirical study replication.

There is a considerable body of work related to feature location. Dit et al. recently provided an excellent systematic survey of this work [10]. The resulting taxonomy classifies the literature along nine key dimensions including type. The three dominate types are dynamic, static, and textual.

Dynamic feature location relies on run-time information. In contrast, static feature location does not. Instead, it statically analyzes the dependencies and structure of the code. Finally, textual feature location exploits the natural language found in source code comments and identifiers to establish a mapping between a textual description of a feature and the relevant parts of the source code.

Some techniques focus on a single type while others combine multiple types. For example, dynamic analysis generally yields good recall, while textual analysis has good precision. Thus there is value in their combination. In combination with static approaches, textual analysis helps reduce the over-estimation to which static analysis is prone [10].

Any technique based wholly or partially on textual analysis necessarily depends on the vocabulary contained in the source code and thus should benefit from vocabulary normalization. For example, Shepherd et al., leverage the use of verbs and their direct objects (nouns) in source code [34]. Expansion of abbreviations and acronyms is a necessary pre-cursor to extracting the part-of-speech information required by this NLP technique.

In their survey, Dit et al. note that “with textual analysis, preprocessing options, such as stemming and stop word removal, are commonly used, but their effect on feature location has not been fully studied” [10]. Perhaps more important than these traditional preprocessing steps is the
application of vocabulary normalization. The results presented in Section IV clearly illustrate the value normalization brings to the feature-location task. Similar value could be expected with other textual-based techniques including hybrid techniques.

Recently there have been two experiments that evaluate the impact that human modifications to identifiers have on IR-based feature locators. One study by Dit et al. [9] compared human splitting to conservative splitting and found that no significant gains were realized by using perfect splitting. This case study was performed using two Java programs. In support of this result, the same conclusion can be drawn for at least one C program, Mosaic, based on the data presented in Section V where no significant gains were realized by an aggressive splitter. This reinforces the notion that expansion is necessary to improve tool performance.

The second study by Abebe et al. [1] automatically identified lexical bad smells including extreme contraction in identifiers, which were removed by expanding identifiers by hand. Two C++ programs were used in the case study. In the program where significant improvement to feature location was realized by fixing bad smells, more than 40% of these changes involved expansion. This is another indication that vocabulary normalization is important to the success of IR-based feature locators.

Turning to the first of three alternate approaches to vocabulary normalization, the need to normalize vocabulary in support of IR-based tools was first noted by Feild et al. [12], [21]. This early work incorporated a limited form of wildcard expansion (described in Section II): words from source code and then a dictionary are searched using a pattern based on the abbreviation. When there was a single match, it was returned as the expansion for a soft term. Despite the algorithm simply failing when zero or more than one match occurred, it correctly expand 40% of a sample of 64 identifiers. In comparison, Normalize uses co-occurrence information about possible expansions and the context to help choose between multiple possible expansions.

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-zA-Z][a-zA-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. The challenge with incorporating wider sources of information is filtering out irrelevant vocabulary. Normalize exploits co-occurrence information to filter out such information.

The second improvement 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 key 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 lent 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.

Finally, replication is an important part of empirical software engineering. Unfortunately, such replication is often quite challenging. Some of the issues are alluded to in Section IV. A more comprehensive treatment of the challenges in replicating empirical study is presented by Bowes et al. [3]. They conclude that researchers need to do a better job of providing sufficient detail to support third-party replication. There is a trend in this direction. For example the feature-location survey of Dit et al. includes supporting online benchmarks in the hopes of supporting future investigation, including future replication [10].

VII. Conclusion

This work contributes to the state of the art in understanding the impact of vocabulary normalization. Even the smallest improvement of 20% for the Original Set suggests the benefit of vocabulary normalization. Because normalization is able to recover key domain terms that were shrouded in invented vocabulary, it is able to greatly improve the ranks of relevant documents in an IR environment. This improvement is most pronounced for shorter, more natural, queries. In support of previous findings [9], the empirical data also demonstrates that splitting alone, whether conservative or aggressive, does not expose all of the important vocabulary present in the source code.

VIII. Acknowledgements

Thanks to David Christo for his help with the experimentation and to Andrian Marcus for his assistance replicating the original study. Support for this work was provided by NSF grant CCF 0916081.

REFERENCES


