Normalizing Source Code Vocabulary

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

Keywords: source code analysis tools, information retrieval, program comprehension

Abstract

Information Retrieval (IR) based tools complement traditional static and dynamic analysis tools by exploiting the natural language found within a program’s text. Tools incorporating IR have tackled problems, such as feature location, that previously required considerable human effort. However, to reap the full benefit of IR-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 consistent. Vocabulary normalization aligns the vocabulary found in source code with that found in other software artifacts. Normalization both splits an identifier into its constituent parts and expands each part into a full dictionary word to match vocabulary in other artifacts.

An algorithm for normalization is presented. Its current implementation incorporates a greatly improved splitter that exploits a collection of resources including several dictionaries, frequency distributions derived from the corpus of programs, and co-occurrence data. Empirical study of this new splitter, GenTest, on almost 8000 identifiers finds that it correctly splits 82%, outperforming the current state-of-the-art. A preliminary experiment with the normalization algorithm finds it improving the FLAT $^3$ feature locator’s scores of relevant code from 0.60 to 0.95 on a scale from 0 to 1.

1 Introduction

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 [27]. Recent research has found that software systems contain significant and useful natural language information [2, 9, 23, 24, 29, 35, 36]. Furthermore, exploiting this information complements existing tools and techniques based on the structure of a program (for example, tools exploiting data and control dependence information). Many of the challenges faced by software engineers as they attempt to recover information from existing software projects can be effectively addressed using IR techniques applied to the unstructured text found in the source code and its associated documents [33]. 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 [31], and performing feature location [35, 36].

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 and requirements documents). 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 [21]. 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 [27]. A previous study found a wealth of natural language information within the source code, but a major stumbling block in applying IR techniques to code is the disparate vocabulary used in source code, especially when compared to external documentation [23]. Thus, there is a need for vocabulary normalization if the full benefit of applying IR techniques is to be realized. Normalization will bring the vocabulary of the code and documentation in line with each other, thus making it more appropriate for consumption by IR-based tools.

This paper makes the following contributions:

1. It presents an algorithm, Normalize, for vocabulary normalization. This algorithm involves two key tasks: splitting and expansion.

2. Efficient implementation of Normalize requires a fast precise splitter. The paper describes in detail the construction of a new splitting algorithm, GenTest, which accurately splits an identifier into its constituent parts.

3. Finally, it empirically evaluates GenTest, finding that it’s 82% accuracy improves on the state of the art.
In the remainder of this paper, Section 2 describes background material. Then Section 3 presents the normalization algorithm. After considering prior splitting approaches in Section 4, the building and evaluation of GenTest in considered in Sections 5 and 6. Finally, Sections 7, 8 and 9 present related work, future challenges, and a summary of the paper.

2 Background

This section first introduces some necessary terminology used to describe identifiers and then the two subtasks that make up normalization. When applying IR to software engineering artifacts, the first step is to break the input text up 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 SE problems because most IR algorithms make the (often implicit) assumption that the same or similar collections of words is used to describe a concept in all documents.

In the following, 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 either 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 characters in camel-cased identifiers can lead to two possible splits – one where all uppercase characters make up a hard-word and one where the final uppercase character is part of the succeeding hard-word. Samurai’s mixed case algorithm [13] produces hard-words with sufficient accuracy, even in the presence of multiple adjacent uppercase characters, that the problem of discovering hard splits in camel-cased identifiers is considered solved and not considered further in this paper.

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 [10, 2, 33, 1]. Such restricted 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 … .” [36]

Their need for such manual preprocessing illustrates both of the 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, it still presents difficulties to an IR-based tool because IR techniques tend not to equate f and feature nor loc and location. To establish the correct link, the soft-words need to be expanded mapping f to feature and loc to location. 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 the normalization algorithm and then explains how the algorithm is used as a preprocessing step to improve IR-based tools. As described above, normalization has two tasks: splitting an identifier into soft-words and expansion of those soft-words to associate a meaning (e.g., a dictionary word) with each. While there are advantages to carrying the two steps concurrently as they can inform each other, for ease of presentation the two are described sequentially.

Task 1 is to separate hard-words into soft-words composed of character sequences that represent words, abbreviations, or acronyms. A generate and test algorithm, named GenTest, is used to accomplish Task 1. The generation part of algorithm is simple as it generates all possible splitting. While this generates an exponential number of splittings, most hard-words 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 describe the quality of the split. The construction of this function is described in Section 6.

Task 2 assigns a meaning to each soft-word. There are two cases. The first is the easy case where the soft-word is a dictionary word and has a well-established meaning that coincides with the use of the word in the source code. 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 a combination of wild-card expansion [22] and ideas taken from machine translation. For example, given the abbreviation h0r1z, the code and the external documentation are searched for strings matching “h*o*r*i*z*”, where a “*” represents an
arbitrary sequence of letters. 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 the word horizontal.

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 documentation can be mined for phrases that may be expansions for acronyms. For example, “cms” appears within many of ghostscript’s identifiers. A phrase finder [16] uncovers the text Color Management System (CMS) in the ghostscript documentation, which is a likely expansion for the identifier.

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 [25], is the planned basis for the Normalize algorithm.

The following description of Normalize is a declarative statement of the algorithm. Because, the set Splits(id) (defined below) includes an exponential number of possible splits, the efficient implementation of the algorithm requires some care. Fortunately, in practice only a handful of potential splits needs to be considered. For example, in the empirical valuation the GenTest splitter ranks the correct split in the top ten over 99% of the time. Sections 5 and 6 describe the engineering of an effective splitter used in the implementation of Normalize.

The formalization of the function Normalize identifies the best expansion over all possible splits. It uses Splits(id) to denote the set of all possible splits of identifier id. A split s ∈ Splits(id) is composed of a sequence of soft-words s_1 s_2 ... s_n. Finally, for soft-word s_i, E(s_i) denotes the set of distinct expansions e_{i,1}, e_{i,2}, ..., e_{i,m} for s_i. The heart of the algorithm is a similarity metric computed from co-occurrence data. This data is used because it has proven useful in resolving translation ambiguity. In other words, the normalization relies on the fact that expanded soft-words should be found co-located in the documentation or in general text. For the general text, a data set of over a trillion words extracted by Google and distributed by the Linguistic Data Consortium [8] is used. In the algorithm the similarity between two expansions, sim(e_1, e_2) is the probability of e_1 and e_2 co-occurring in a five word window in the Google data set.

**Part 1** For a selected splitting s = s_1 s_2 ... s_n

1. For each expansion e_{i,j} ∈ E(s_i), define the similarity score between e_{i,j} and the other soft-words of s, s_k (k ≠ i) as the sum of the similarities between e_{i,j} and the elements of E(s_k)

   \[ sim(e_{i,j}, s_k) = \sum_{e \in E(s_k)} sim(e_{i,j}, e) \]

2. Define the cohesion for e_{i,j} relative to s as

   \[ cohesion(e_{i,j}, s) = \log \left( \sum_{s_k \in s \neq s_i} sim(e_{i,j}, s_k) \right) \]

3. For each s_i ∈ s, score(s_i) is the cohesion of the expansion e_{i,j} ∈ E(s_i) having the maximal cohesion

   \[ score(s_i) = \max_{e_{i,j} \in E(s_i)} \left[ cohesion(e_{i,j}, s) \right] \]

**Part 2** Finally, for the identifier id, Normalize(id) identifies the split with the highest cumulative score

\[ Normalize(id) = \max_{s \in Splits(id)} \left[ \sum_{s_i \in s} score(s_i) \right] \]

As an example of how this algorithm works consider the hard-word str-len where there are two possible splits: Splits(str-len) = \{ str-rlen, str-len \}. Let E(st) = \{ stop, string, set \}, E(rlen) = \{ riflemen \}, E(str) = \{ steer, string \}, and E(len) = \{ lender, length \} as the possible expansions (translations) of each soft-word. Similarity scores are computed using the Google data set.

\[
\begin{align*}
\text{sim}(\text{stop}, \text{riflemen}) &= 9.95857 \times 10^{-8} \\
\text{sim}(\text{string}, \text{riflemen}) &= 9.95857 \times 10^{-8} \\
\text{sim}(\text{set}, \text{riflemen}) &= 9.95857 \times 10^{-8} \\
\text{sim}(\text{steer}, \text{lender}) &= 0 \\
\text{sim}(\text{steer}, \text{length}) &= 9.95857 \times 10^{-8} \\
\text{sim}(\text{string}, \text{lender}) &= 0 \\
\text{sim}(\text{string}, \text{length}) &= 0.00262389
\end{align*}
\]

With these similarity scores, \( \text{sim}(e_{i,j}, s_k) \) can be computed. If \( e_{i,j} = \text{string} \) and \( s_k = \text{len} \) then

\[
\begin{align*}
\text{sim}(\text{string}, \text{len}) &= \text{sim}(\text{string}, \text{lender}) + \text{sim}(\text{string}, \text{length})
\end{align*}
\]

in Part 1a. The computation for cohesion sums over all the other soft-words besides str. In this case there is one other soft-word so,

\[ \text{cohesion}(\text{string}, \text{str-len}) = \log(\text{sim}(\text{string}, \text{len})) \]

in Part 1b. The maximal cohesion for Part 1c is chosen from among the possible expansions. Given

\[
\begin{align*}
\text{cohesion}(\text{string, str-len}) &= -5.9431 \\
\text{cohesion}(\text{steer, str-len}) &= -16.1222 \quad \text{score(str)} = -5.9431 \text{ using string as the expansion.}
\end{align*}
\]

Normalize then identifies the split with the highest overall score sum. In the example, str-len produces the sum \(-11.8862\), while for str-rlen the sum is \(-31.1459\). Therefore, in Part 2 Normalize(str-len) selects str-len using the expansions string and length because it has the maximal score sum.
4 Prior Splitting Approaches

The GenTest algorithm which produces possible splits for Normalize builds on ideas presented in two prior algorithms: Greedy and Samurai. This section introduces these two algorithms. The Greedy Algorithm [15] relies on a dictionary to determine where to insert a split in a hard-word. The algorithm identifies in the hard-word the longest prefix or suffix that is in the dictionary. Once a dictionary word is discovered, it is set aside as a soft-word, and the remaining characters are recursively searched for additional soft-words. Since both the prefix search and suffix search are invoked on the remaining characters, the search returning the higher ratio of soft-words found in the dictionary to the total number of soft-words is used. The process ends when the remainder is a dictionary word or contains no dictionary words.

The dictionary used by the Greedy Algorithm is composed of three groups of words. The dominant group is natural language words. For example, the initial experiments used the publicly available dictionary that accompanies ispell Version 3.1.20. The second group augments the dictionary with common abbreviations (e.g., alt for altitude) and programming abbreviations (e.g., txt for text). The final group is made up of programming-language specific words including keywords (e.g., while), predefined identifiers (e.g., NULL), library function and variable names (e.g., strcpy and errno), and all identifiers that consist of a single character.

The second splitting algorithm, Samurai [13], scores potential splits using the frequencies of the occurrence of strings from two sources: those appearing in the program being analyzed and those appearing in a large corpus of programs. The strings used by Samurai are the hard-words extracted from the identifiers of both sources, together with the words found in the comments of both sources. The algorithm builds two tables that map each string to the number of times that the string occurs as a hard-word or in a comment. One table is constructed from the program being analyzed, yielding the program-specific frequency table, progFreq, and the other from the entire corpus, yielding the global frequency table, globalFreq. In addition, hand-built prefix and suffix lists are used to prevent certain strings from appearing as soft-words. Prefixes include “afro”, “co”, and “peri” while suffixes include “aholic”, “eous”, and “tropy”.

The two tables are used in the following string scoring function:

\[ \text{progFreq}(s, p) + \frac{\text{globalFreq}(s)}{\log(\text{AllStrsFreq}(p))} \]

where \( p \) is the program under analysis and \( \text{AllStrsFreq}(p) \) is the total number of strings that occur in \( p \). With the goal of maximizing the score, the algorithm compares the score for the entire hard-word and those for all possible splits of the hard-word into two soft-words subject to two constraints:

- first neither the left soft-word is on a list of prefixes nor the right soft-word is on a list of suffixes. The second constraint is that there be overwhelming evidence in favor of the split. Such evidence exists if the square root of the two soft-words’ scores is greater than that of the unsplit hard-word.
- If the first constraint holds, but only the left soft-word has a sufficiently high score, then the right soft-word is recursively split, but not vice versa.

5 GenTest Scoring Metrics

This section presents the metrics used by GenTest. In contrast to previous splitting algorithms, rather than identifying chunks of a hard-word as acceptable and then attempting to split the remainder of the hard-word, GenTest uses a generate and test strategy in which all possible splits are materialized and then scored (i.e., tested). When used only for splitting, the one with the highest score is selected. When used by Normalize, a ranked list of high-scoring splits is used to prioritize the expansions considered.

Although there are an exponential number of possible splits relative to the length of the hard-word, in practice the number of possible splits is manageable because identifiers, and thus their hard-words, tend to be short. For those where the number of splits becomes large, machine learning techniques such as genetic algorithms can provide an efficient way of exploring the search space given an accurate scoring function.

The metrics that GenTest uses aim to characterize a high quality splitting of a hard-word. After describing each metric, motivation for the metric is provided. In Section 6, logistic regression is used to empirically build a model and thus a scoring function that captures the best combination of metrics.

There are three categories of metrics: soft-word characteristics, metrics incorporating external information, and metrics incorporating internal information. Soft-word characteristics are characteristics of the strings produced by the splitting. External information includes dictionaries and other information that is either human engineered or extracted from non-source code sources. Internal information is derived from the source code, either the program itself or a collection of programs.

5.1 Soft-word Characteristics

There are five metrics derived from soft-word characteristics. The first is simply the number of soft-words the hard-word is divided into, number of words. This can be anywhere from 1 to \( n \) where \( n \) is the length of the hard-word. Empirical analysis shows that generally fewer soft-words are better.

The second metric is the average soft-word size, average word length. This is simply the average length of the soft-word strings. In general, longer soft-words are more understandable.
The third metric is the longest soft-word size, *longest word*. This is highly correlated to average soft-word size. It was included because programmers frequently add a short prefix to a well understood soft-word such as when using Hungarian notation.

The fourth metric, *words with vowels*, is the number of soft-words that contain vowels. The value of this metric is higher when there are fewer vowel-free soft-words; however, even in the case of non-words many abbreviations include some vowels. This metric captures that expectation.

The final metric, *single letter count*, is the number of soft-words containing a single letter (i.e., length one). Unlike the other metrics, it is hypothesized that smaller values for this metric will lead to better splits.

### 5.2 External Information

Eight metrics make use of external information such as dictionaries to calculate metric values. Three of the eight are derived from counting the number of soft-words found in the dictionary. Two different dictionaries are used. One is Debians’s *wamerican* (6-2) which consists of a concatenation of Kevin Atkinson’s SCOWL word list sizes 10 thru 50 [3]. This includes 98,569 entries. The other dictionary has 479,625 entries and is distributed in /usr/share/dict with Red Hat 4.1.2-14. The first dictionary is referred to as the ‘small dictionary’ and the second as the ‘large dictionary’. The small dictionary is a subset of the large dictionary. One metric, *small dictionary match count*, is the number of soft-words found in the small dictionary. The second metric, *large dictionary match count*, is the number of soft-words found in the large dictionary. The third metric, *large dictionary match len 3 count*, is the number of soft-words that have a length of three or more and are found in the large dictionary.

The fourth metric, *programming word count*, uses the same list of programming specific words employed in Greedy Splitting. In this case, the number of soft-words that are found on the list is counted. Since the list is programming language specific, the programming language of the file where the soft-word occurs is taken into consideration when counting.

The fifth metric, *dictionary expansion count*, counts the number of soft-words that are not on the programming specific list and that can be found in the large dictionary or expanded into a word in the small dictionary. To expand a soft-word, the soft-word is first stemmed using Krovetz’s morphological stemmer [20]. Then, provided that the soft-word has more than one character, the dictionary is searched for a wild-card expansion, as described in Section 3. If one or more matches are found for a soft-word, then that soft-word is counted as having a dictionary expansion.

The final metrics are computed using co-occurrence information developed by Google and provided by the Linguistic Data Consortium. The data set provides the number of times a series of 5 words (5-grams) is observed on web pages crawled by Google as of 2006. Co-occurrence is computed for words $w_1$ and $w_2$ by counting the number of times $w_1$ appears in a 5-gram with $w_2$ divided by the number of 5-grams $w_2$ occurs in.

For example, let $w_1$ = *linguistics* and $w_2$ = *Google*. Assume the following 5-grams:

<table>
<thead>
<tr>
<th>5-gram</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>about the future of Google</td>
<td>101</td>
</tr>
<tr>
<td>exactly the same as Google</td>
<td>87</td>
</tr>
<tr>
<td>Google Scholar People in Linguistics</td>
<td>93</td>
</tr>
</tbody>
</table>

The Co-occurrence(*linguistics*, *Google*) = $\frac{93}{101+87+93} = 0.331$, which means that roughly one-third of the time that *Google occurs linguistics* also occurs.

There are three metrics computed from the co-occurrence information. *Co-occurrence* is the average co-occurrence of each successive pair of soft-words in the hard-word. Through inspection it was discovered that many two character ‘words’ in this set are a result of faulty optical character recognition (OCR). These small words caused excessive splitting. *Co-occurrence len 3* only includes pairs of soft-words where both words in the pair are of length three or longer. The final metric, *co-occurrence when combined not word*, also attempts to cope with the OCR problem by ignoring pairs whose combined characters make up a word.

### 5.3 Internal Information

There are five internal metrics. The first metric, *in source*, tallies the number of soft-words that meet one of the following conditions: (1) the characters form an acronym for a phrase found in the source, or (2) the soft-word or an expansion of the soft-word is a dictionary word appearing in the comments or the code. Phrases are identified by running the comments and multi-word-identifiers through a phrase finder [16]. Here, the first letter of each word in the phrase is used to build an acronym. If a soft-word matches an acronym exactly, then the soft-word is part of the tally. For instance, the phrase finder extracts the phrase *Color Management System* from *ghostscript* documentation. This phrase matches the soft-word *cms*. This metric counts higher quality expansions than the *dictionary expansion count* by including the subset of the dictionary that is found in the source. This works well when abbreviations are defined elsewhere in the source code.

The final four metrics are rooted in the frequency tables used in the Samurai algorithm. However, instead of straight frequency counts, these metrics use normalized frequencies – the frequency of a string divided by the total number of strings observed. A normalized frequency can be interpreted as the probability of encountering a particular string
in the program or collection. Assuming that these probabilities are independent and disjoint (a common assumption in IR even though it is generally false), the probability of observing the union of a set of words is obtained by summing their individual probabilities, while the probability of observing intersection of the words from the set is obtained by computing the product of their probabilities.

For these metrics the words in the unions and intersections include only soft-words that have three or more characters. Short soft-words are given a very small or no probability since their counts are likely to be inflated by homonymy. Short soft-words have an artificially high probability because they can expand into so many different concepts. The probability of the union of the soft-words is computed using both the program normalized frequencies and the global or collection normalized frequencies leading to the union-program probability and union-global probability metrics. The probability of the intersection of the soft-words is computed using both the program normalized frequencies and the global normalized frequencies leading to the final metrics, intersection-program probability and intersection-global probability.

6 Building and Evaluating GenTest

GenTest combines the metrics from Section 5 using statistical analysis. This section first describes the oracle data used in this analysis and the building of the model from the oracle data set. It then compares GenTest’s results with the two algorithms presented in Section 4 and finally provides a brief discussion of threats to validity.

6.1 Oracle Data Set

The oracle data set identifies the correct splitting for each identifier. It was generated by having four programmers hand split overlapping subsets of a random sample of four thousand identifiers. The random sample was drawn from a source base that includes 186 programs containing 26MLoC of C, 15MLoC of C++, and 7MLoC of Java. The total of almost 50MLoC includes almost 3 million identifier instances, of which 746,345 are unique. These are composed of 104,278 unique hard-words. To produce the oracle data, 4,000 identifiers were randomly chosen from the 746,345.

The 4,000 identifiers contain 2,180 unique hard words. Because the same hard-word can generate different program specific metric values, each program was searched for identifiers containing that hard-word. This increased the size of the oracle to 8,455 identifiers. Finally, hard-words containing fewer than three characters or more than twelve characters were excluded. One and two character hard-words almost never require splitting while those longer than the twelve characters tend to be composed of juxtaposed (English) words where a simpler splitter, such as the Greedy algorithm [15], is more efficient and accurate. In total, 7,941 hard-words formed the oracle. To avoid over-fitting the data, the oracle set was randomly divided into two halves: an estimation set, used for model construction, and a validation set.

Oracle identifiers come from three different programming languages. The bulk, 67%, are from C programs. The data set is 26% C++ identifiers and 7% comes from Java programs. This roughly reflects the source code base. The most frequently occurring hard-word in the data set is val with 50 occurrences, whereas accfragref is among the 1,141 hard-words that only occur once.

6.2 Model Construction and Quality

Statistical analysis by SAS version 9.1 was used to generate a model of correctly-split identifiers. Because the response variable is binary (whether a particular split is right or wrong), logistic regression is used to model the association between the response variable and the explanatory variables. The resulting model, a weighted combination of the statistically significant metrics, becomes the scoring function for potential splittings.

The first step in model construction is to generate the metric values for all possible splits of each oracle hardword. However, because no hard-word required more than three splits, those with four of more splits were ignored.

Because each identifier still has multiple splits, repeated measures are present in the data; thus, a generalized linear mixed model (GLMM) [32]) is appropriate to handle the within identifier correlations caused by repeated measures. However, fitting such a model produced very small (essentially zero) variance estimates, suggesting that, the more sophisticated GLMM analysis is not needed. Thus, multiple logistic regression is used to fit the models to obtain predicted probabilities of a correct prediction.

When modeling the data, two of the metrics were eliminated: co-occurrence when combined not word, which was not significant ($p > 0.05$) and intersection-global probability, which while statistically significant, but had a very small effect and was highly correlated with intersection-program probability.

The resulting scoring function appears as Equation (2) in Figure 1. Statistically, it is a very good model of the data. The area under the ROC curve is 0.984 (the c-statistic), which is in the “excellent” range. The high percent concordant value of 97.8% indicates that the correct splitting has a higher predicted probability than incorrect splits 97.8% of the time. Somer’s D and the Gamma statistics also indicate an almost perfect association.

The following discussion details how the metrics contribute to the overall score. First, the coefficients of Equation (1) are either positive or negative. For negative coefficients smaller values produce higher scores. The opposite is true for positive coefficients. For example, a higher score comes with fewer words (i.e., a lower value of num-
Logistic Regression Parameter Estimates

<table>
<thead>
<tr>
<th>Explanatory Variable</th>
<th>Min</th>
<th>Max</th>
<th>Error</th>
<th>$\chi^2$</th>
<th>Pr &gt; $\chi^2$</th>
<th>Odds-Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of words</td>
<td>1</td>
<td>4</td>
<td>0.1620</td>
<td>1497.5287</td>
<td>&lt; .0001</td>
<td>0.002</td>
</tr>
<tr>
<td>programming word count</td>
<td>0</td>
<td>2</td>
<td>0.1847</td>
<td>534.9597</td>
<td>&lt; .0001</td>
<td>71.738</td>
</tr>
<tr>
<td>in source</td>
<td>0</td>
<td>4</td>
<td>0.1118</td>
<td>353.4142</td>
<td>&lt; .0001</td>
<td>8.179</td>
</tr>
<tr>
<td>small dictionary match count</td>
<td>0</td>
<td>4</td>
<td>0.0774</td>
<td>578.7864</td>
<td>&lt; .0001</td>
<td>6.442</td>
</tr>
<tr>
<td>large dictionary match count</td>
<td>0</td>
<td>4</td>
<td>0.1071</td>
<td>17.6977</td>
<td>&lt; .0001</td>
<td>0.637</td>
</tr>
<tr>
<td>large dictionary match len 3 count</td>
<td>0</td>
<td>3</td>
<td>0.1097</td>
<td>379.1848</td>
<td>&lt; .0001</td>
<td>8.465</td>
</tr>
<tr>
<td>single letter count</td>
<td>0</td>
<td>4</td>
<td>0.1392</td>
<td>46.7953</td>
<td>&lt; .0001</td>
<td>2.592</td>
</tr>
<tr>
<td>dictionary expansion count</td>
<td>0</td>
<td>4</td>
<td>0.0975</td>
<td>331.1915</td>
<td>&lt; .0001</td>
<td>5.897</td>
</tr>
<tr>
<td>co-occurrence</td>
<td>0</td>
<td>4</td>
<td>0.0543</td>
<td>285.6</td>
<td>&lt; 0.001</td>
<td></td>
</tr>
<tr>
<td>co-occurrence len 3</td>
<td>0</td>
<td>4</td>
<td>0.0543</td>
<td>286.7</td>
<td>0.0123</td>
<td></td>
</tr>
<tr>
<td>average word size</td>
<td>1</td>
<td>12</td>
<td>0.0603</td>
<td>248.7042</td>
<td>&lt; .0001</td>
<td>0.386</td>
</tr>
<tr>
<td>longest word</td>
<td>1</td>
<td>12</td>
<td>0.0491</td>
<td>178.237</td>
<td>&lt; .0001</td>
<td>1.928</td>
</tr>
<tr>
<td>words with vowels</td>
<td>0</td>
<td>4</td>
<td>0.0716</td>
<td>103.6284</td>
<td>&lt; .0001</td>
<td>0.482</td>
</tr>
<tr>
<td>union-program probability</td>
<td>0</td>
<td>4</td>
<td>0.0395</td>
<td>23.7065</td>
<td>&lt; .0001</td>
<td></td>
</tr>
<tr>
<td>intersection-program probability</td>
<td>0</td>
<td>4</td>
<td>0.0020</td>
<td>465.0384</td>
<td>&lt; .0001</td>
<td>0.946</td>
</tr>
<tr>
<td>union-global probability</td>
<td>0</td>
<td>4</td>
<td>0.0294</td>
<td>33.4389</td>
<td>&lt; .0001</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Statistical information concerning each of the explanatory variables in the model.

$$\eta = 8.2271$$
- 6.27 number_of_words
+ 4.27 programming_word_count
+ 2.10 in_source
+ 1.86 small_dictionary_match_count
- 0.45 large_dictionary_match_count
+ 2.14 large_dictionary_match_len_3_count
+ 0.95 single_letter_count
+ 1.77 dictionary_expansion_count
- 714.9 co-occurrence
+ 829.2 co-occurrence_len_3
- 0.95 average_word_length
+ 0.66 longest_word
- 0.73 words_with_vowels
+ 85.58 union-program_probability
- 0.06 intersection-program_probability
+ 377.0 union-global_probability

$$\text{Score} = \frac{e^\eta}{1 + e^\eta}$$

Figure 1. GenTest’s Scoring Function

$\text{ber}\_\text{of}\_\text{words}$; however, more programming-words (i.e., a higher value of $\text{programming}\_\text{word}\_\text{count}$) is better. In essence, the model says that all things being equal fewer soft-words are better, but if there are more soft-words, it is best that they be on the programming list. Second, to fully appreciate the impact a metric has, it is important to consider the range of values that were observed as shown in Table 1 by the minimum and maximum columns. This limit bounds the impact that a metric can have on the overall value of Equation (1). For example, the most that union-program probability contributes to Equations (1) is 3.38 ($85.5753 \times 0.03951$), whereas number of words potentially has a greater impact as it largest value while negative has a magnitude of 25.08.

There are several interesting features in Equation (1). The number of words is the most influential single variable. However, the combined program probabilities can make a very high positive contribution to the score. In fact, many of the similar metrics have opposite signs, $\text{co-occurrence}$ and $\text{co-occurrence}\_\text{len}\_3$ for instance, because they are highly correlated, members of such a pair have a dampening effect on each other. Both are left in the model as each also brings something unique to the model. In the case of the two co-occurrences, $\text{co-occurrence}\_\text{len}\_3$ is always less than or equal to $\text{co-occurrence}$; therefore, the highest scores will result when the two are equal. Also, although the coefficients for these variables are large, the maximum values are small and thus the pair never has a large impact on the value of Equation (1).

Finally, Table 1 includes some statistical information relating to how good the logistic regression parameter estimates are and the odds-ratio. Since the model is a logistic regression, the value produced by the score can be interpreted as the odds that the split is the correct split. The standard error is the usual estimate of variability/uncertainty in the parameter estimate. The Wald $\chi^2$ statistic, which is
6.3 Results and Comparison

of success for a unit increase in the explanatory variable. The parameter estimate is negative the odds-ratio will be less than one and the effect of the explanatory variable (i.e., metric). If the parameter estimate is positive the odds of success for a unit increase in the explanatory variable on the response is a decrease in the odds-ratio will be greater than one, and the effect of the explanatory variable on the response is an increase in the odds-ratio over-split at nearly the same rate that an additional split 58% of the time. The Greedy algorithm attributes the inaccuracies. An inaccurate split is the result of a missed split. Samurai more evenly distributes the inaccuracies. Over 76% of the time there is an inaccurate split, the inaccuracy is caused by a missed split. Samurai more evenly distributes the inaccuracies. An inaccurate split is the result of an additional split 58% of the time. The Greedy algorithm over-split at nearly the same rate that GenTest under-splits, 77% of the time. Thus, if under-splitting is more desirable than over-splitting, GenTest also performs better when it is inaccurate.

Table 2. Accuracy of the different algorithms overall and by programming language

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Overall</th>
<th>C</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenTest (val)</td>
<td>81.82%</td>
<td>80.91%</td>
<td>82.66%</td>
<td>87.64%</td>
</tr>
<tr>
<td>GenTest (est)</td>
<td>82.07%</td>
<td>80.81%</td>
<td>82.54%</td>
<td>92.01%</td>
</tr>
<tr>
<td>Greedy</td>
<td>64.45%</td>
<td>65.57%</td>
<td>60.17%</td>
<td>69.88%</td>
</tr>
<tr>
<td>Samurai</td>
<td>70.32%</td>
<td>69.98%</td>
<td>67.72%</td>
<td>83.54%</td>
</tr>
</tbody>
</table>

The goal of the approach is to identify matchings between identifier substrings and dictionary words. The identifier is considered a signal of unknown meaning described by the feature vector $x_1, x_2, ..., x_N$. Each dictionary word is then used as a second (known) signal described by the feature vector $y_1, y_2, ..., y_M$. The algorithm performs a dynamic

6.4 Threats to Validity

Most of the threats to the validity of this work are typical for this kind of statistical modeling. For example, the external validity is threatened by the selection of only certain identifiers for the oracle set. This sample may not be representative of identifiers in general, although given the randomness of the sample and the similar performance on the estimation and validation sets, it is likely that the sample is representative of (open source) software. One threat is particular to this experiment: following Anquetil and Lethbridge [1], vocabulary normalization assumes that software engineers are trying to give identifiers meaningful names (although they may have failed in the attempt).

7 Related Work

This section briefly considers the broader category of IR-based tools giving one example, and then considers previously published splitting and expansion algorithms. Many of the changes 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 [36, 35], reverse engineering [17], impact analysis [9], software clustering metrics [29], software libraries [31], developer identification [24], and software comprehension [10]. Of these application categories, over half are rather 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 [29]. 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.

Two splitting algorithms were presented in Section 4. A third was inspired by speech recognition techniques [26]. The goal of the approach is to identify matchings between identifier substrings and dictionary words. The identifier is considered a signal of unknown meaning described by the feature vector $x_1, x_2, ..., x_N$. Each dictionary word is then used as a second (known) signal described by the feature vector $y_1, y_2, ..., y_M$. The algorithm performs a dynamic
time warping (DTW) of \( x \) and \( y \) to find the optimal match between the two vectors. The “time warp” part of the search allows \( N \) and \( M \) to differ and in the splitting domain allows abbreviations to be accounted for. The optimal match is made by computing local distances and then choosing matches that minimize the overall distance using dynamic programming.

When comparing the four algorithms, Samurai’s need for overwhelming evidence produces fewer over-splits than the Greedy Algorithm, but more than GenTest. The DTW algorithm is highly dependent on the dictionary used. The presented results [26] appear dependent on a small focused dictionary. When specialized dictionaries are not available, the technique may be prohibitively dependent on a small focused dictionary. A head-to-head comparison with the DTW algorithm is left to future work as it will require careful planning to correctly understand and evaluate the dictionary’s impact on the algorithm’s performance.

Finally, three expansion algorithms are considered here. The DTW algorithm provides hints for expansion by identifying abbreviations of dictionary words. When the algorithm uses an abbreviation, the algorithm can map the abbreviation back to the original word. However, this feature of the algorithm is not evaluated and currently has no way to choose among multiple words that produce the same abbreviation. The Scoped Approach [18] finds possible expansions in dictionary words found in the source code and documentation. Expansions are associated with soft-words based on regular expression patterns. When multiple expansions are possible, higher frequencies of occurring in the source code are favored. The final expansion approach uses wildcard expansion [22] as described in Section 3. The source code is used as the initial source of expansions and a dictionary is used as a secondary resource. Soft-words are associated with an expansion when there is only one possible expansion.

8 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 [11, 28]. Such a probabilistic approach is well suited for automatic abbreviation and acronym detection. For example, \( n \)-gram language models [30], built from \( n \) sequential letters, have been successfully used in speech recognition software [28]. In essence, relative entropy identifies unusually frequent sequences of characters. These are likely to be meaningful in a particular document. For example, “\texttt{cms}” appears within many of \texttt{ghostscript}’s identifiers. Since it has a high relative entropy, it is identified as an acronym used in the program. Adding a relative entropy metric should benefit GenTest. Furthermore, combined with a phrase finder [16] it should allow the automatic expansion of acronyms.

This paper focuses on the splitting aspect of normalization. Expansion can follow splitting by taking the chosen split as input and attempting to expand any non-dictionary soft words. However, the two aspects are expected to perform better after integration. For example, GenTest can produce a ranked-list of the top \( X \) splits. If expansion works better with one of these ten, then it is the preferred split. In support of integrating the two GenTest ranks the correct split in the top ten over 99% of the time.

Once the normalization implementation is in place, empirical investigation of its impact on existing tools is planned. Tools and techniques to be experimented with will be drawn from the collection of problems in software engineering to which IR has been applied [5, 6, 14]. Vocabulary normalization is expected to dramatically improve existing (and future) IR-based tools. As an illustration of the impact that Normalize can have considers on the FLAT*3 [34] 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 paired GenTest with a simple expansion algorithm. The results showed a dramatic improvement in FLAT*3 performance. An example that highlights the benefits of expansion, considers the search for “account number”. Before normalization this search returns files containing \( \texttt{accountNum} \) having a confidence (cosine similarity) of 0.60. After normalization, which replaces \texttt{accountNum} with \texttt{accountNumber}, the confidence jumps to over 0.95. Similar improvement is expected from other IR-based SE tools.

9 Summary

IR-based techniques complement techniques grounded in structural (compiler) based analysis, which presently dominate the field. Popular examples include Prevent [12], Klockwork K7 [19], and historically lint [7]. Further examples can be found in a recent survey of the past, present, and future of Source Code Analysis [4].

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 and that used in the documentation. 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. The splitting algorithm leverages a collection of metrics to correctly split a hard-word 82% of the time. Preliminary experiments with the second phase, finds that it works well. Further evolution of the algorithms for both
phases and continued empirical study are expected to show the continued improvement and benefit of vocabulary normalization.

10 Acknowledgments

Thanks to Justin Overfelt and Austin Wheeler for their input on the application of Normalize. Special thanks to the anonymous referees for their suggestions, which contributed to a substantial improvement to the paper. Support for this work was provided by NSF grant CCF 0916081.

References


