A Comparison of Stemmers on Source Code Identifiers for Software Search

Andrew Wiese, Valerie Ho, Emily Hill
Department of Computer Science
Montclair State University
Montclair, NJ, USA
{*wiesea1, hov2, hillem*}*@mail.montclair.edu*

***Abstract*—As the popularity of text-based source code analysis grows, the use of stemmers to strip suffixes has increased. Stemmers have been used to more accurately determine relevance between a keyword query and methods in source code for search, exploration, and bug localization. In this paper, we investigate which traditional stemmers perform best on the domain of software, specifically, Java source code. We compare the stemmers using two case studies: a comparative analysis of the unified word classes in terms of accuracy and completeness, as well as an investigation into the effectiveness of stemming for software search. Our results indicate that relative stemmer effectiveness varies with a software engineering tool such as search, justifying further research into this area.**

***Keywords*-stemming; textual analysis of source code identifiers; source code search;**

I. Introduction and Background

Stemming is the process of stripping affixes, such as prefixes and suffixes, from words to form a stem. Stemming is often applied to words in Information Retrieval (IR) systems so that words with almost the same meaning, but superficial spelling differences, are grouped together as the same concept. This process of grouping related words together is also known as *word conflation*. For example, if a user searches for the query 'adding auctions' in an IR system, the user would most likely be interested in documents containing the words 'add' and 'auction'. By appropriate use of stemming, IR systems can recognize that 'add' and 'adding', as well as 'auction' and 'auctions', map to the same ideas, despite surface differences in spelling.

Stemmers suffer from two sources of errors: understemming and overstemming. Understemming occurs when a stemmer does not produce the same stem for all the words in the same concept (i.e., does not *conflate*, or group, words of the same concept). Understemming can reduce the number of relevant results returned in a search (i.e., reduce recall), since fewer results will be considered as relevant to the query. In contrast, overstemming occurs when a stemmer gives the same stem for words with different meanings [1], and can increase the number of irrelevant results returned by search (i.e., reduce precision). For example, an aggressive stemmer might stem 'business' to 'busy' or 'university' to 'universe'. Stemmers have been categorized by their strength [2], based on whether they are *light* and favor understemming, or *heavy* and favor overstemming.

A. Light Stemmers

The Porter stemmer [3] is widely used in the IR community for its simplicity and efficiency. The Porter stemmer forms a stem by iteratively applying a sequence of rules to strip common English suffixes. As a result of its speed and simplicity, the Porter stemmer produces some inaccurate stems. For example, the related words 'add' and 'adding' are not conflated to the same stem. Porter later created a definitive implementation of his original 1980 stemmer, with a few minor rule improvements ($< 5\%$) [4]. We refer to the second Porter algorithm as Snowball.

To address the shortcomings of the purely rule-based Porter stemmer, Krovetz developed a derivational stemmer that uses word morphology (i.e., using the word's internal structure) and a hand-tuned dictionary of words and exceptions [1]. We refer to this stemmer as KStem. KStem was found to outperform Porter's, especially for collections of shorter documents (≈40-60 words) [1].

B. Heavier Stemmers

Like Porter, Lovins [5] and Paice [6] developed stemmers that attempt to find the longest matching rule before stemming. In contrast to Porter, the Lovins and Paice stemmers are heavy, tending to overstem. Empirical studies have shown that both Lovins and Paice are heavier than Porter's [2], [7], although the studies disagree as to which stemmer is the heaviest of all. In the remainder of this work, we focus on the newer Paice stemmer as a representative heavy rule-based stemmer.

We also created a heavy morphological stemmer specialized for software using the two-level morphological parser PC-Kimmo [8]. We select between possible parses using word frequency information from a corpus of 9,000+ open source Java programs. Using this technique, we identified stems for 21,332 words found in the Java corpus. MStem is available at http://msuweb.montclair.edu/~hillem/stemming.

C. Stemming Source Code

The challenge with existing stemmers is that they have been developed and tuned for use with the English used in natural language documents. However, the restricted vocabulary of software source code can present different challenges. For instance, in object-oriented programming, classes are

sometimes used to encapsulate actions, such as a 'player' class for a 'play' action or a 'compiler' for a 'compile' action. Thus the action's verb, such as play, is nominalized into a noun, like player. Search performance can decrease if the stemmer does not stem the action and its nominalization to the same word. Light stemmers like Porter are less likely to conflate words that have different parts of speech.

In this paper, we investigate which stemmers perform best on the domain of software using two initial case studies of Java source code. First, we compare the differences between the word classes produced by stemmers on the top 100 most frequently occurring words in a set of over 9,000 open source Java programs, analyzed in terms of accuracy and completeness by two human evaluators. Second, we evaluate the use of stemming in searching software. Our results indicate that relative stemmer effectiveness varies with a software engineering tool such as search.

II. Comparison of Stemmer Word Classes

We compare the conflated word classes generated by 4 common IR stemmers (Porter, Snowball, KStem, and Paice) and one specialized for software (MStem).

A. Study Design

We used the 100 most frequently occurring words found in identifiers from a set of over 9,000 open source Java programs downloaded from source forge. For each word, we created its equivalent word class by calculating the list of words that each stemmer conflates to the same stem, yielding over 1900 unique words. For this initial study, we compare the conflated word classes in terms of two binary measures: accuracy and completeness. The word class of a stemmer is considered to be *accurate* if it contains no unrelated words to the concept. For example, a word class for 'element' containing 'else' is inaccurate. We consider a word class to be *complete* if it contains all the conceptually related words from the union of word classes for all the stemmers. For example, a word class for 'element' that is missing 'elements' is incomplete. Prior evaluations have had humans exhaustively annotate alphabetized word lists [7], precluding words like 'buy' and 'bought' or 'length' and 'long' from being annotated as equivalent. Instead, we rely on the diversity and greediness of the stemmers used in the study to aid our humans in determining completeness.

In general, judging whether two words reflect the same concept is a challenging problem that is difficult for humans to objectively determine. As with prior stemmer evaluation, we assume that careful human judgement is a reasonable approximation of reality [7]. However, there are some words that have multiple senses, or where the conceptually similar words depends on the context of the search query or target software engineering application. To keep these ambiguous cases separate in our analysis, the humans annotated these words as *context sensitive*.

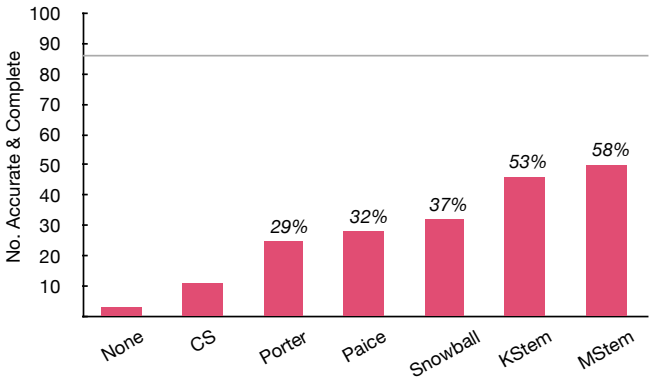

Figure 1. Summary of annotations by word. Includes the number of accurate & complete words for each stemmer, the number of context sensitive (CS), and where no stemmer is both accurate & complete (None).

Two humans familiar with Java programming annotated the conflated word classes for each stemmer. The evaluators independently annotated the word classes for each stemmer in terms of accuracy and completeness, and then met in person to determine agreement. The annotators both agreed to base accuracy and completeness on words that occurred at least 150 times in 15 or more programs. These are very low thresholds, considering the most frequently occurring word, 'the', occurred over 22 million times and the most common word, 'public', occurred in over 9500 programs. Words that were difficult to reach agreement or had multiple valid interpretations were annotated as 'context sensitive'.

In addition to frequency of occurrence, the annotators agreed that stemmers which group words that are unrelated ('public' and 'publisher'), unusual ('and' and 'anded'), or foreign language forms ('list' and 'listar') are inaccurate. In general, the annotators agreed to exclude alternate forms of abbreviated words, marking 'comming' for 'com', 'xmls' for 'xml', and 'ios' for 'io' as inaccurate. Out of 100 words, there were only 2 exceptions to this rule, by considering the plural forms for 'id' and 'int' to be valid.

B. Results and Analysis

Figure 1 presents a summary of accuracy and completeness annotations by word. There were 14/100 words for which no stemmer was clearly accurate and complete. Each stemmer bar is annotated with the percent of the remaining 86 words for which the stemmer is accurate and complete. For example, out of the 86 words, MStem generated word classes that are accurate and complete for 58% of the words, whereas Paice was accurate and complete for just 32% of the words. The gray horizontal line plots the 86 word maximum.

As shown in Figure 1, there were only 3/100 words where no one stemmer was clearly accurate and complete, and only 11 context sensitive words. For example, 'add' is a word with multiple senses and thus which words are related will depend on the sense being used in context. Consider searching for the arithmetic sense of 'add', which would be related to words such as 'adder', 'addition', and 'addable'. In

Table I
STEMMER WORD CLASS COMPARISONS FOR 4 EXAMPLES (UNDERLINED WORDS ARE IN THE WORD CLASSES FOR ALL STEMMERS)

Word (A & C)	Stemmer	Word Class
element (MStem)	Porter	element, elemental, elemente, elements
	Snwbl	element, elemental, elemente, elements
	KStem	element
	MStem	element, elemental, elements
	Paice	el, ela, ele, element, elemental, elementary, elemente, elementen, elements, elen, eles, eli, elif, elise, elist, ell, elle, ellen, eller, els, else, elseif, elses, elsif
import (Kstem)	Porter	import, importable, importance, important, imported, importer, importers, importing, imports
	Snowbl	import, importable, importance, important, importantly, imported, importer, importers, importing, imports
	KStem	import, importable, imported, importer, importers, importing, imports
	MStem	import, importable, importance, important, importantly, imported, importer, importers, importing, imports
	Paice	import, importable, importance, important, importantly, importar, imported, importer, importers, importing, imports
add (CS)	Porter	add, adde, addes, adds
	Snwbl	add, adde, addes, adds
	KStem	add, addable, added, addes, adding, adds
	MStem	add, addable, added, adder, adding, addition, additional, additionally, additions, additive, additivity, adds
	Paice	ad, ada, add, addable, adde, added, adder, addes, adding, adds, ade, ads
name (None)	Porter	name, named, namely, names, naming
	Snwbl	name, named, namely, names, naming
	KStem	name, nameable, named, namer, names, naming
	MStem	name, named, nameless, namely, namer, names, naming, surname
	Paice	name, nameable, namely, names

contrast, the sense of 'add' being used to join something to a list is not typically related to 'addition'. The word classes for 'add', as well as 3 other examples, are shown in Table I.

Overall, the annotators found the morphological parsers MStem and KStem to be the most accurate. The results of these two subjects indicate that morphology may be more important than degree of under- or overstemming, since MStem is a heavy stemmer and KStem light. MStem was the only accurate and complete stemmer for 12 of the words, whereas KStem was accurate and complete for 11. In contrast, the rule-based stemmers Porter and Snowball were uniquely accurate and complete stemmers for 2 words, and Paice 6. Of the rule-based stemmers, light Snowball has a clear advantage over light Porter and heavy Paice overall.

As expected with heavy stemmers, MStem and Paice both tend to overstem, although for different reasons. MStem frequently stems across different parts of speech, which generally leads to increased completeness. However, occasionally this tendency conflates words that do not represent the same concept, as in conflating the adjective 'true' with the adverb 'truly' and noun 'truth'. In contrast, Paice frequently conflates unrelated words together, such as 'element' with 'else' and 'static' with 'state', 'statement', 'station', 'stationary', 'statistic', and 'status'.

The annotators observed a difference between the morphological stemmers (MStem and KStem) and the rule-based stemmers (Porter, Paice, and Snowball), which frequently and inaccurately associated non-words or foreign language words. For example, all 3 rule-based stemmers conflated 'method' with french 'methode' and 'methodes'; 'panel' with Spanish 'paneles'; and 'any' with non-words 'anys' and, in the case of Porter and Snowball, 'ani'. MStem and KStem were less prone to these errors because MStem uses word frequencies to eliminate unlikely stems, and KStem uses an English dictionary.

C. Threats to Validity

Because the words were selected exclusively from Java programs, these results may not generalize to all programming languages. MStem was trained on the same set of 9,000+ Java programs that were used to create the 100 most frequent word set annotated by the human evaluators. Due to the large size of the entire word set (over 700,000 words), it is unlikely that MStem was over-trained on the subset of 100 words. Since completeness is based on the union of word classes created by the stemmers, the observations may not generalize to all morphological and rule-based stemmers. Because determining accuracy and completeness can be ambiguous, we limited this threat by separating out the 'context sensitive' examples in our analysis.

III. EFFECT OF STEMMING ON SOURCE CODE SEARCH

In this section, we compare the effect of using Porter, Snowball, KStem, Paice, and MStem with no stemming (None) on searching source code.

A. Study Design

To compare the effect of stemming on software search, we use the common tf-idf scoring function [9] to score a method's relevance to the query. Tf-idf multiplies two component scores together: term frequency (tf) and inverse document frequency (idf). The intuition behind tf is that the more frequently a word occurs in a method, the more relevant the method is to the query. In contrast, idf dampens the tf by how frequently the word occurs in the code base. Because we recalculate idf values for each program and stemmer combination, the tf-idf scores can widely vary between stemmers that are heavy and light.

We use 8 of 9 concerns and queries from a previous source code search study of 18 developers [10]. For one concern no subject was able to formulate a query returning any relevant results, leaving us with 8 concerns. For each concern, 6 developers formulated queries, totaling 48 queries, 29 of which are unique. The concerns are mapped at the method

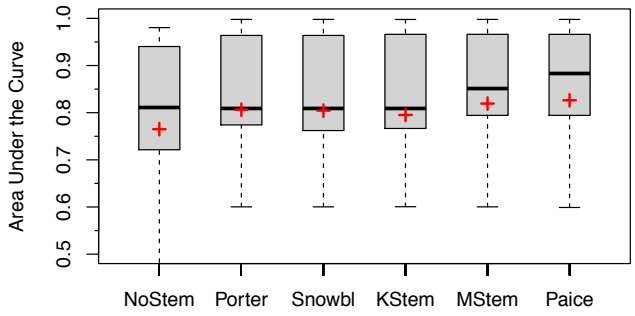

Figure 2. Area Under the Curve for Stemmer Variants.

level, and contain between 5–12 methods each. The programs contain 23–75 KLOC, with 1500–4500 methods [10].

Search effectiveness is typically calculated at a particular *threshold*, such as top 5, 10, or 20 results. When a threshold is selected, all results returned by the search above the threshold are considered to be relevant, and the remainder irrelevant. However, our interest is to evaluate the use of stemming independent of search threshold. Thus, we use the area under the ROC curve (AUC) [9] to quantify how quickly recall increases with the threshold. The AUC is calculated by plotting the increase in the number of true positives by the number of results returned, finding the area under that plot, and normalizing by the maximum possible area. AUC values vary between 0.5, which is equivalent to randomly ordering the search results, and 1.0, which represents returning all the relevant results before any irrelevant ones.

B. Results and Analysis

Figure 2 presents a box and whisker plot capturing the overall AUC results for each stemmer across all concerns. The shaded box represents the inner 50% of the data, the middle line represents the median, and the plus represents the mean. As expected, Figure 2 shows that stemming ranks relevant results more highly than irrelevant ones. However, the median and third quartiles for no stemming, Porter, Snowball, and KStem share very similar ranges. This implies that stemming helps most for the challenging queries, and that with an adequate query, little stemming is necessary.

Although all stemmers performed similarly on average, the medians in Figure 2 indicate that the greedier stemmers, MStem and Paice, perform the best with tf-idf. This is in contrast to the human stemmer evaluation in Section II. Paice outperformed all other stemmers for 2 concerns, one where Paice overtemmed and one where Paice understemmed. In both cases, Paice's mistakes happened to increase search results. For example, consider the 'textfield report' concern. The query 'texfield report element' is not ideal, causing irrelevant results to be ranked higher than relevant ones for most stemmers. Paice's overstemming 'element' with 'else' caused the idf to drop so low that Paice's results for 'textfield report element' match the ideal 'textfield report' query. More concerns are necessary to verify the interaction between heavy stemmers and source code search.

C. Threats to Validity

For this study, we have selected a small set of concerns used in a previous concern location study [10], and the search results may not generalize to all concerns. We have endeavored to use realistic search results by using human-created queries. Because our focus is on Java programs, these results may not generalize to searching programs written in other programming languages.

IV. Conclusions and Future Work

In this paper, we investigated the use of stemmers on the domain of software, specifically, Java source code. Based on the initial results, morphological stemmers appear to be more accurate and complete than rule-based stemmers. In contrast, heavy stemmers such as MStem and Paice, which tend to overstem, appear to be more effective in searching source code. Our initial results show that relative stemmer effectiveness can be affected by a software engineering tool such as search. In general, stemming produces relevant results more consistently than not stemming. We observed that the heavy and morphological stemmers tended to outperform the light rule-based stemmers, Porter and Snowball, for search. Future experiments will further investigate stemming source code words from both a human and software search perspective, investigate stemmer effectiveness on less frequent and more domain-specific words, and evaluate the use of stemming on other software engineering problems and artifacts.

References

[1] R. Krovetz, "Viewing morphology as an inference process," in *Proc. Int'l ACM SIGIR Conf.*, 1993.

[2] W. B. Frakes and C. J. Fox, "Strength and similarity of affix removal stemming algorithms," *SIGIR Forum*, v 37, n 1, 2003.

[3] M. Porter, "An algorithm for suffix stripping," *Program*, v 14, n 3, pp. 130–137, 1980.

[4] M. F. Porter, "Snowball: A language for stemming algorithms," 2001. http://snowball.tartarus.org/texts/introduction.html

[5] J. Lovins, "Development of a stemming algorithm," *Mechanical Translation & Computational Linguistics*, v 11, 1968.

[6] C. D. Paice, "Another stemmer," *SIGIR Forum*, v 24, 1990.

[7] ——, "An evaluation method for stemming algorithms," in *Proc. Int'l ACM SIGIR Conf.*, 1994.

[8] E. L. Antworth, *PC-KIMMO: a two-level processor for morphological analysis*, 1990. http://www.sil.org/pckimmo/

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[10] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2007.