Published in Working Conference on Source Code Analysis and Manipulation (SCAM 2012)

On the Use of Stemming for Concern Location and Bug Localization in Java

Emily Hill
Department of Computer Science
Montclair State University
Montclair, NJ, USA
hillem@mail.montclair.edu

Shivani Rao, Avinash Kak
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{*sgrao, kak*}*@purdue.edu*

***Abstract*—As the popularity of text-based source code search and analysis grows, the use of stemmers to strip suffixes has increased. Although widely investigated in the information retrieval community, the comparative effectiveness of stemmers in the domain of software is relatively unknown. In this paper, we investigate which of the well-known stemmers perform best in the domain of Java software for concern location and bug localization. For these two problems, we evaluate the use of stemming on over 500 search tasks for six different Java applications. Using MAP and Rank Measure, we conducted an overall qualitative study and a query-by-query quantitative study of the impact of stemming on retrieval effectiveness. As one might expect, our contribution demonstrates that how stemming affects retrieval performance is mediated by other factors, such as the use of tf-idf to filter commonly occurring terms and the precise nature of the queries. Specifically, we find that the extent to which stemming improves the retrieval performance relates to the degree of natural language content in a query.**

***Keywords*-stemming; textual analysis of source code identifiers; concern location; bug localization**

I. Introduction

As the popularity of text-based source code analysis has grown, so has the use of stemmers to more accurately determine the relevance of keyword queries to source-code artifacts [1], [2], [3], [4], [5], [6], [7], [8]. Although widely investigated in the information retrieval (IR) community [9], [10], [11], [12], [13], [14], [15], the effectiveness of stemmers in the domain of software is relatively unknown [16].

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.

Although stemmers in IR systems have had mixed success in practice [11], [12], their use in software retrieval needs to be investigated independently due to the differences between natural language and source code identifier vocabulary. For example, in natural language documents, a relevant term is likely to appear in many forms. While in English we may consider the phrases 'exploring a program' and 'program exploration' to be identical in meaning, a compiler does not consider the identifiers ExploringAProgram and ProgramExploration to refer to the same entity. Thus, because the same identifier must be used to indicate all definitions and uses of the same program entity, programmers are forced to use a smaller vocabulary. When applied to a smaller vocabulary, stemming could have a greater effect on the overall retrieval effectiveness.

In prior work, Wiese, et al. [16] investigated the use of stemmers on Java source code by comparing human judgements on 100 frequently occurring words and by searching for eight concern location search tasks. We go beyond this preliminary study by evaluating stemmer effectiveness on two different types of software search tasks—concern location and bug localization—and increasing the number of search tasks from eight to over 500.

The two types of search tasks deal with different types of queries. The queries that are used for concept location are keyword-style natural language queries. In contrast, the queries for bug localization typically contain code snippets in addition to natural language. The two different styles of queries allow us to explore the relationship between stemming and type of query. As we show, while the nature of the query plays an important role in the retrieval, there are several other factors that also affect the stemmer effectiveness. To study these effects, we adapted Hull's stemmer evaluation method [12] to isolate "interesting" cases of queries for qualitative analysis. Section III-A presents this adaptation of Hull's method while Sections IV and V present the interesting queries and the corresponding analysis.

II. Background

A. A Brief Introduction to Stemming

Stemmers suffer from two sources 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 relevant to the query. For example, by not conflating 'add' and 'adding' with the same stem, a search for 'add' would miss relevant results containing 'adding'. In contrast, overstemming occurs when a stemmer gives the same stem for words with different meanings [13], 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 [10], based on whether they are *light* and favor understemming, or *heavy* and favor overstemming.

In this study, we use the Porter [17] stemmer and its variant, Snowball [18] as light, algorithmic stemmers. Algorithmic stemmers iteratively apply a sequence of rules to strip common English suffixes, and are widely used in the IR community for simplicity and efficiency. Although the light Porter stemmer is the most widely used, it is preceded by the heavier Lovins stemmer [19] and followed by the heavier Paice [20] stemmer. Empirical studies have shown that both Lovins and Paice are heavier than Porter's [10], [15], 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 algorithmic stemmer.

In contrast to algorithmic stemmers, morphological stemmers use the internal structure of a word to derive stems. Krovetz developed KStem, which uses word morphology and a dictionary of words and exceptions [13]. KStem was found to outperform Porter's, especially for document collections where the document length was short (≈40-60 words on average) [13]. For a heavy morphological alternative to light KStem, we use MStem, a heavy morphological stemmer specifically tuned for software [16]. MStem is available at http://msuweb.montclair.edu/~hillem/stemming. More detailed information about these stemmers and the challenges of applying stemmers to source code can be found in prior work [16].

B. Prior Stemmer Evaluation

Since 1981, a number of IR studies have investigated the impact of stemming on retrieval effectiveness, with varying results. All but one of the studies of stemmer effectiveness have concluded that although stemming may not have a large impact overall, it can result in large improvements in retrieval effectiveness for specific queries. From the perspective of a developer when faced with a particular query, stemming may provide a critical advantage in successfully achieving the developer's search task.

Although Lennon, et al. [14] found little difference in retrieval effectiveness among stemmers—whether heavy (Lovins) or light (Porter)—they concluded that in some instances stemming can significantly increase retrieval effectiveness, but does not decrease it. Lennon, et al.'s results were based on a single document collection, Cranfield. In contrast, Harman [11] found stemming showed no significant improvement in retrieval effectiveness when comparing Lovins, Porter, and a simple s-removal baseline, and concluded that stemming harms as often as it helps based on results from 3 document collections: Cranfield, Medlars, and CACM. Krovetz [13] contradicted these findings by concluding that stemming results in significant improvements in retrieval performance, and the greatest improvements are seen when the documents are fairly short (tens of words per document rather than hundreds or thousands). Krovetz compared Porter, Snowball, and KStem on 4 document collections: CACM, TIME, NLP, and WEST.

More recently, Hull [12] compared Lovins, Porter, s-removal, and morphological stemmers (not KStem) and found that although absolute stemmer improvement is just 1–3% overall, stemming can have a large impact for individual queries. Fautsch and Savoy [9] later replicated Hull's study with additional probabilistic IR techniques and more queries, and found that most stemmers produce similar results, yielding around a 7% improvement overall.

Other stemmer evaluations investigate stemmer strength, concluding that Paice and Lovins are heavier than Porter [15], [10]. A preliminary study by Wiese, et al. used both human judgement and retrieval effectiveness to compare stemmers on Java source code. Although the human judgements found the morphological stemmers (MStem, KStem) to be more accurate and complete than the algorithmic stemmers (Snowball, Paice, Porter), retrieval effectiveness on 8 search tasks favored the heavier stemmers (Paice, MStem) over the light ones (KStem, Snowball, Porter). These conflicting results of relative stemmer effectiveness on Java source code inspired the current study, which evaluates stemmer retrieval effectiveness on over 500 search tasks.

III. A Framework for Stemmer Comparison

To investigate the effectiveness of stemmers on Java software search, we perform two studies: concern location and bug localization. In this section we summarize the similarities in experimental design and analysis between both studies. In Sections V and IV we provide further details about the search tasks and methodology for each study.

In both studies, we compare four traditional IR stemmers (Porter, Snowball, KStem, and Paice) and one specialized for software (MStem). We selected our stemmers based on whether they are heavy or light, morphological or algorithmic, or specialized for the domain of software. Both Porter and Snowball are light algorithmic stemmers, KStem is a light morphological stemmer, Paice is a heavy rule-based stemmer, and MStem is a heavy morphological stemmer specialized for Java source code. We used implementations of Porter and Snowball from Porter's website, http://snowball.tartarus.org/. For Paice, we used the perl

implementation at http://www.comp.lancs.ac.uk/computing/research/stemming/. We used the lucene implementation of KStem, which can be found at http://ciir.cs.umass.edu/downloads/files/KStem.jar. For MStem, we used the word list at http://msuweb.montclair.edu/~hillem/stemming.

A. Analysis Methodology

Search effectiveness is typically calculated in terms of *precision*, which measures the proportion of relevant documents retrieved out of all of the documents returned, and *recall*, which measures the proportion of relevant documents retrieved out of all the relevant documents for a given query. Precision and recall are typically calculated at a particular rank. In traditional IR, rank-based precision/recall measurements are converted into the calculation of Average Precision (AP) [21], [22], which is the area under the precision-recall curve that results from the individual precision and recall values at different ranks. When AP is averaged over the set of all queries, we get what is known as the Mean Average Precision (MAP), which signifies the proportion of the relevant documents that will be retrieved on average for a given query. A MAP of 0.2 means that, on average, a retrieval for a query returns one relevant document for every 5 documents retrieved. The higher the MAP, the more effective the retrieval algorithm.

In addition to calculating raw MAP values for each stemmer and no stemming, we also calculate for each query the difference in MAP values between each stemmer and no stemming, which we call *MAP Difference*. For comparing the values of MAP for the different stemmers, we use the ANOVA F-test to analyze if any of the MAP values (i.e., means) of the techniques are significantly different. Subsequently, we apply Tukey's Honest Significant Differences (HSD) test to evaluate the degree and direction of the mean differences. This statistical analysis procedure has been recommended for IR experiments to avoid the multiple comparisons problem [23].

Although we test for statistically different MAP values, traditional IR experiments have not found stemming to significantly affect retrieval effectiveness overall [12]. Instead, these prior IR studies have found that stemming can considerably improve retrievals for certain types of queries. In order to gain a deeper understanding of the impact of stemming on each query, we adapt the detailed evaluation methodology from Hull's work [12] and start by partitioning the queries into the following subsets:

- Q_+: the set of queries where stemming yields improvements over not stemming, regardless of stemmer
- $Q_=$: the set of queries where stemming yields the same results as not stemming
- Q_-: the set of queries where stemming hurts performance, regardless of stemmer
- Q_{vary}: the remaining queries, where stemming can improve, hurt, or have no effect on retrieval effectiveness, depending on which stemmer is used

Quartile	Short	Medium	M291	Long	AOC	Rhino
Max	22	2328	5790	8392	5	9
Q3	10	265.5	331.5	975.5	2	5
Median	8	133	172	421	1	4
Q1	6	71.5	68	269.8	1	3
Min	2	2	2	50	1	1
Mean	8.571	247.0	319.6	769.9	1.792	4.009

Table I
DISTRIBUTION OF NUMBER OF QUERY WORDS FOR SUBJECTS

In addition to subjecting the MAP values to the ANOVA F-test, we also compare the retrieval performance using what Hull [12] refers to as *Rank Measures*. Hull suggests analyzing the variability in the ranks of the retrieved documents for the same query but for different stemmers. Subjecting the Rank Measures to the ANOVA F-test reveals the presence of any significant differences between the ranks on a per-query basis. High F-values indicate high variance in the retrieved ranks for the different stemmers and can be used to isolate "interesting" examples for further qualitative analysis.

B. Threats to Validity

Because our focus is on Java software, these results may not generalize to searching programs written in other programming languages. Although we have compared stemmers across multiple Java programs, the results may not hold for other Java programs. The results of the bug localization study may not generalize to other programs, since they are only for the AspectJ. Another potential threat to validity is poorly split identifiers. Although we have applied the same camel-case identifier splitting technique to all software artifacts and queries, poorly split identifiers can reduce the impact of stemmers on search. For example, a search for "adding a file" could miss the relevant word "add" in the poorly split identifier addfile. We do not expect identifier splitting to significantly reduce the impact of stemming in our study, because there are relatively few Java identifiers that are difficult to split as compared to those splittable by camel case [24], [25].

We have employed the VSM model for the concern location problem and the Unigram model for the bug localization problem. Our results may not be generalizable to other retrieval models. For example, the Unigram model was chosen for the bug localization search task since it is known to give the best performance [28] for the iBugs test collection used here. This makes Unigram a good *baseline* model for performance comparisons between stemming and no stemming, but may not generalize to other IR techniques.

IV. EFFECT OF STEMMING ON BUG LOCALIZATION

Bug localization is the process of finding the program elements (methods and fields) that directly relate to a given bug. As a search task, IR-based bug localization takes as input a textual description of a bug and outputs a ranked list

of results [26], [7]. In this section, we investigate the impact of stemming on an IR-based approach to bug localization.

A. Subject Bugs and Queries

Evaluating the stemmers for bug localization requires a set of bug reports and related source code artifacts. In this study, we use the iBUGS [27] dataset. iBUGS was created by mining five years of version history of the ASPECTJ software and its bug-tracking system. iBUGS contains 75 KLOC and over 350 bugs, 291 of which relate to program elements in Java source files. These 291 bugs form the ground-truths (i.e., gold set) for our study.

Bug reports contain multiple sources of textual information that can be used as queries. Each bug typically has a title, or *short description*, and one or more comment lines in the main body of the bug description. To investigate the effect of query length on stemmer effectiveness [12], we use three different types of queries: the first comment line in the bug description as used in the original iBugs dataset (medium descriptions), the short descriptions (i.e., titles), and the entire bug description text, which we refer to as the long description. Because not all bugs contain short and long descriptions, these sets contain just 126 bugs. Thus, we have 4 categories of queries: Short (126 bugs), Medium (126 bugs), Long (126 bugs), and M291 (medium length queries but all of the 291 bugs). The mean, median, min, and max length of the query in each category is tabulated in Table I.

B. Methodology

In a prior study of IR techniques for bug localization, Rao and Kak [28] demonstrated that simple retrieval models, in particular the Vector Space Model and the Unigram Model, outperform the more complex models such as those based on the Latent Semantic Analysis and Latent Dirichlet Allocation. For the purposes of this study, we use the Unigram Model (UM) because of its prior effectiveness [28] and its scalability.

The Unigram Model (UM) represents each document with a probability distribution [29]. This probability distribution is smoothed using a collection-wide distribution of terms, $p_c(w)$. Thus, the smoothed Unigram representation of the m^{th} document can be expressed as:

$$\begin{aligned} p_m(w) &= (1-\mu)\frac{tf_m(w)}{\sum_w tf_m(w)} + \mu p_c(w) \\ p_c(w) &= \frac{\sum_m tf_m(w)}{\sum_w \sum_m tf_m(w)} \end{aligned} \quad (1)$$

where $tf_m(w)$ represents the term frequency of the w^{th} word in the document m. Because queries are subject to the same processing and smoothing steps as documents, a query q can be represented by the probability distribution:

$$p_q(w) = (1-\mu)\frac{tf_q(w)}{\sum_w tf_q(w)} + \mu p_c(w) \quad (2)$$

Query	None	Paice	KStem	MStem	Snowball	Porter
Long	0.2236	0.1955	0.2175	0.2194	0.2151	0.2130
Medium	0.1419	0.1446	0.1499	0.1559	0.1387	0.1492
M291	0.1549	0.1482	0.1529	0.1512	0.1465	0.1525
Short	0.1133	0.1049	0.1167	0.1138	0.1137	0.1106

Table II
MEAN MAP SCORES FOR IBUGS

Query	Paice	KStem	MStem	Snowball	Porter
Long	-0.02806	-0.006090	-0.004199	-0.008529	-0.01055
Medium	0.002638	0.007970	0.01400	-0.003243	0.007280
M291	-0.006703	-0.002056	-0.003738	-0.008391	-0.002462
Short	-0.008479	0.003343	0.0004492	0.0003093	-0.002758

Table III
MEAN MAP DIFFERENCE SCORES FOR IBUGS

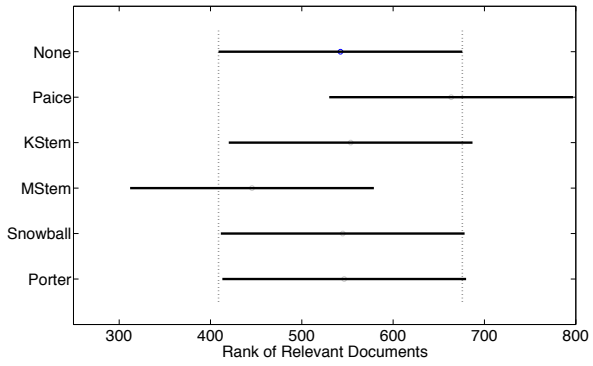
With both the documents and the query being represented as $|\mathcal{V}|$ dimensional probability vectors, we use KL divergence to determine the "match" between a query and a document. These similarity values are then ranked in decreasing order to create a ranked list of documents for each query. Finally, we calculate the MAP values and we perform the statistical analyses described in Section III-A.

C. Results and Analysis

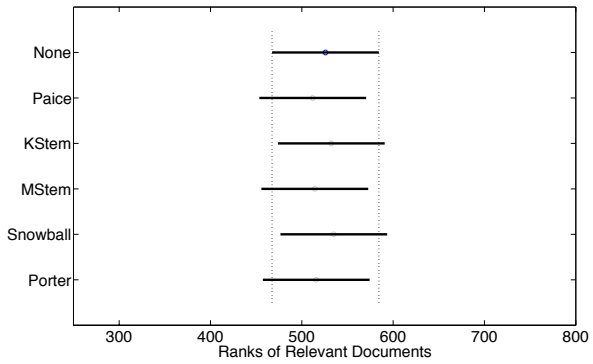
Tables II and III show the MAP and MAP Difference results of each stemmer for the three different types of queries. As can be seen from the second column of Table II, retrieval effectiveness of no stemming improves in direct proportion to query length. Since most information in the comments are code snippets containing class names, they are less impacted by stemming as compared to shorter queries. Hence we see no improvement when using stemming for long queries. However, stemming impacts shorter queries, although Paice tends to reduce retrieval effectiveness.

To further explain these results, we investigate specific cases where we can observe the relative strengths and weaknesses of individual stemmers by comparing the query subsets Q_+, $Q_=$, Q_-, and Q_{vary}, as defined in Section III-A. Table IV shows the number of queries in each subset. Figures 1(a), 1(b) and 1(c) show the mean and variance rank measure for each type of query in the Q_{vary} set. The x-axis denotes the ranks of the relevant documents and each horizontal line indicates the variance, centered about the mean. The rank of relevant documents shows more variability among stemmers as compared to Average Precision. Note stemming has a higher impact on shorter queries than medium or long, although none of the differences are statistically significant.

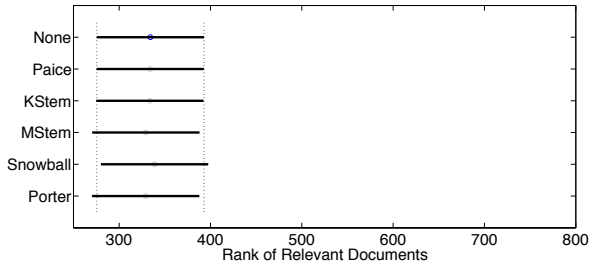
Stemming > No Stemming (Q_+): There are some queries that are loaded with verbs and their variants, and no matter what stemming algorithm is employed, the conflation significantly reduces the number of unique terms in the query and hence improves the weight of each of the conflated terms, which aids in better retrieval. For example, bug



(a) Short Queries



(b) Medium Queries (M291)



(c) Long Queries

Figure 1. Distribution of Ranks for $q \in Q_{vary}$ for iBugs

50776 has terms like *compiler, compiles, compiling, overriding, introduction, differing, throws, exception, exceptions, redefining,* and *invocation*, all of which can be pruned to a more concise form. Hence these are the queries that contain a lot of verbs and their variants. Another example is bug 129525, which contains terms like *load, loader, loading, message, messages, information, intention, exception, execution, circularity, dump, dumped,* and *dumping*.

Stemming < No Stemming (Q_-): In spite of the presence of verbs and their variants, some otherwise well-performing queries are negatively impacted by stemming. This occurs when the relevant files contain the exact form of the words in the query and conflation only adds noise by

Query Type	Q_+	Q_-	$Q_=$	Q_{vary}
Short	29	46	4	47
Medium	25	34	6	61
M291	53	92	12	134
Long	25	36	5	60

Table IV
NUMBER OF QUERIES WHERE STEMMING HELPS (Q_+), HURTS (Q_-), HAS NO EFFECT ($Q_=$), AND WHERE PERFORMANCE VARIES (Q_{vary}).

retrieving irrelevant documents. An example is bug 34951, where the relevant file and the query both contain *compiling, aspectjrt,* and *aspectjrt.jar*. Stemming *compiling* → *compile* allows unrelated documents containing terms like *compilers, compilation* to out-score the relevant documents containing unique terms like *aspectjrt and aspectjrt.jar*. Another example is bug 128699, where the terms *annotation, changed, declarations,* and *dominates* are present in the relevant file in the exact form. In general, the change in rank is typically small. For example, in the case of bug 128699, the rank of the relevant document is pushed from 5 to the range 7–9.

Stemming = No Stemming ($Q_=$): There are a few queries for which the retrieval performance is unaffected by stemming due to few words being subject to stemming in the query and relevant documents. For example, in bug 49250 the terms that are conflated are not found in the relevant documents, so stemming has no effect.

Stemmer-dependent (Q_{vary}): Although the above three sets give useful insights on the applicability of stemming for bug localization, queries in the Q_{vary} set are the "interesting" cases where the relative strengths and weaknesses of the stemmers are evident. From the rank-based F-tests for the queries in Q_{vary}, we find the ordering in Table V holds.

Paice and MStem > other stemmers: Paice and MStem outperform other stemmers in some cases because they are heavy. For example, bug 29934 contains the term *pointer*. The source files relevant to this bug contain variants of the word *pointer* such as *points* and *point*. While other stemmers fail to conflate *pointer* to *point*, Paice and MStem are able to find more matches between the relevant document and the query. Another example is *configuration* → *config* for bug 109016. The source files often contain the term *config* instead of *configuration*. While other stemmers conflate *configuration* to *configur*, Paice greedily, stems it to *config*, increasing the degree of commonality between the query and the relevant document.

Paice < other stemmers: However, Paice's greediness can also reduce retrieval effectiveness. For example, Paice incorrectly conflates *inter* to *int*. Similarly, Paice conflates *outjar* → *outj* and *ajde* → *ajd*, which reduces the utility of the specific identifier *outjar*. Other greedy conflations that are potentially harmful for retrieval are: *accept*→*accieve*, *compare* → *comp*, *after* → *aft*, *actual* → *act*, *only* → *on*. Although Paice is known to be a greedy stemmer prone to over-stemming, it is sometimes unable to catch simple word

Query	Ordering as given by Rank Measure F-tests
Short	MStem > None > Snowball > Porter > KStem > Paice
Medium	Paice > MStem > Porte > None > KStem > Snowball
M291	Paice > MStem > Porter > None > KStem > Snowbal
Long	MStem > KStem, Paice > None > Snowball
AOC	Paice, MStem > KStem, Porter > Snowball > None
Rhino	Paice > MStem, Porter, Snowball > KStem > None

Table V
RELATIVE ORDERING OF STEMMERS ON THE Q_{vary} SET

forms. For example, for bug 70619, 'enter' and 'entry' are not conflated to 'enter'.

MStem > other stemmers: MStem is sensitive to software terms like *exception* and *throw*. For example, for bug 36803, none of the other stemmers are able to conflate *thrown* to *throw*, and so MStem is able to retrieve relevant documents that none of the other stemmers can. Similarly for bug 109016, MStem retains *exception* in its original form, rather than conflating it to *except* as the other stemmers do, and thus boosts the retrieval effectiveness using MStem.

V. EFFECT OF STEMMING ON CONCERN LOCATION

A *concern* is anything software stakeholders consider to be a conceptual unit, such as features, requirements, or implementation mechanisms [30]. Concern location as a search task takes a keyword-style query and searches the code for program elements (methods and fields) that implement the concern, returning a ranked list of results. Concern location is similar to feature location, except that user-observable features are a subset of the concerns in a software system [31]. In this section, we investigate what effect stemming has on the target software engineering application of concern location. We compare the effect of using Porter, Snowball, KStem, Paice, and MStem with no stemming (None).

A. Search Technique

To compare the effect of stemming on software search, we use the common tf-idf scoring function [32] 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.

B. Subject Concerns and Queries

To evaluate stemmers in terms of search, we need a ground-truth (i.e., gold) set of queries and features, or concerns, for which to search. In this study, we use two sets of subject concerns and human-formulated queries: a set of action-oriented concerns from four Java programs, and a larger set of feature-based concerns from the documentation of a single Java program.

Program	Concern	Queries (No. of subjects)
iReport	Insert textfield	add text, create textfield, text field tool, textfield, textfieldReportElement (2)
iReport	Compile report	IReportCompiler, compile, compile report (4)
jBidWatcher	Add auction	add auction (6)
jBidWatcher	Set snipe bid price	add snipe, do you wish to snipe, set snipe, snipe (2), sniping
jBidWatcher	Save auctions list	save, save auction (4), save auctions
javaHMO	Download movie listings	find, get theater, movie (2), theater, view listing
Jajuk	Search music library	search (5), search file
Jajuk	Play audio file	open file, play, play file (2), play track, start

Table VI
ACTION-ORIENTED CONCERNS AND QUERIES

	Number of Software Developers	
No. Years	**Programming Experience**	**Profesional Programming Experience Outside Coursework**
5+ years	5	2
2-5 years	2	2
1-2 years	1	2
< 1 year	–	2

Table VII
SUBJECT DEVELOPER CHARACTERISTICS FOR DOCUMENTATION-BASED CONCERN SET, RHINO

1) Action-oriented concern set: The first set comprises 8 of 9 concerns and queries from a previous concern location study of 18 developers searching for action-oriented concerns [8]. One of the techniques in the study, Google Eclipse Search (GES) [33], uses keyword-style queries suitable for input to tf-idf. For one concern no subject was able to formulate a query returning any relevant results, leaving us with 8 concerns in our study. For each concern, 6 developers interacted with GES to formulate a query, resulting in a total of 48 queries, 29 of which are unique. The concerns are mapped at the method level, and contain between 5–12 methods each. The programs contain 23–75 KLOC, with 1500–4500 methods [8]. Table VI presents the concerns and queries for this set.

2) Documentation-based concern set: The second set of concerns consists of 215 documented features from the 45 KLOC JavaScript/ECMAScript interpreter and compiler, Rhino, from a set of 415 concerns [34]. Each concern maps to a subsection of the documentation, which is used as the concern description. The number of program elements (methods and fields) in each of the 415 concerns varies from 1 to 334. For the purposes of this study, we only consider concerns containing at least 10 program elements and no more than 110 elements, leaving us with 215 concerns.

To obtain human-formulated queries for the concern set, we asked 8 volunteer software developers familiar with

Set	None	Paice	KStem	MStem	Snowball	Porter
AOC	0.2122	0.2429	0.2384	0.2377	0.2279	0.2298
Rhino	0.09301	0.09597	0.09381	0.09212	0.09290	0.09288

Table VIII
MEAN MAP SCORES FOR CONCERN LOCATION

Set	Paice	KStem	MStem	Snowball	Porter
AOC	0.03072	0.02619	0.02548	0.01576	0.01762
Rhino	0.002955	0.0007937	-0.0008919	-0.0001163	-0.0001374

Table IX
MEAN MAP DIFFERENCE SCORES FOR CONCERN LOCATION

Java programming to read the documentation for a subset of 80-81 concerns. The developers had varying levels of programming and industry experience, shown in Table VII. The subjects were asked to formulate a query containing words they thought were relevant to the feature and would be the first query they would type into a search engine such as Google when searching. They could include specific identifiers as keywords if those were listed in the documentation. The developers were randomly assigned blocks of concerns such that 3 different subjects formulated queries for each concern, yielding a total of 645 concern-query combinations.

C. Methodology

Each stemmer (and no stemming) was used with tf-idf to search for each query. Both the queries and the source code documents (methods and fields) are processed using the same identifier splitting and stemming techniques. Formally, given a query q, stemmed query word w, and a method m, we use the following equation to calculate tf-idf:

$$tf - idf(m) = \sum_{w \in q} (1 + ln(tf_m(w))) * ln(idf(w))$$

where $tf_m(w)$ is the term frequency of stemmed word w in the method m, and ln is the natural log. Because idf is recalculated for each stemmer, the tf-idf scores can widely vary between stemmers that are heavy and light. Unlike bug localization, the textual queries for concern location are typically short, just 2 or 4 words on average for our data sets (see Table I).

D. Results and Analysis

Tables VIII and IX show the mean MAP and MAP Difference results for each stemmer and no stemming for the AOC and Rhino concern sets. It should be noted that none of the differences are statistically significant. Table X shows the number of queries where stemming helps (Q_+), hurts (Q_-), has no effect ($Q_=$) and where performance varies (Q_{vary}).

From Table IX we observe that for AOC, stemming ranks relevant results more highly than irrelevant ones. Taking a closer look at the distribution for MAP Difference, we find that Paice, MStem, and KStem outperform no stemming for 75% of the queries, whereas light Porter and Snowball outperform no stemming for just 50% of the queries. In contrast, Tables IX and X illustrates that for Rhino, stemming seems to hurt as often as it helps. From Table IX we see modest MAP Differences, with Paice and KStem offering slight improvements over stemming, while MStem, Snowball, and Porter perform slightly worse. From Table X we observe that stemming hurts (Q_-) or has variable performance (Q_{vary}) for twice as many queries as it helps (Q_+).

Query Type	Q_+	Q_-	$Q_=$	Q_{vary}
AOC	18	9	3	18
Rhino	112	239	70	224

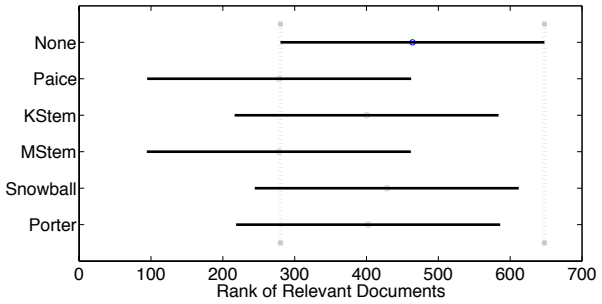
Table X
NUMBER OF QUERIES WHERE STEMMING HELPS (Q_+), HURTS (Q_-), HAS NO EFFECT ($Q_=$), AND WHERE PERFORMANCE VARIES (Q_{vary}).

In terms of aggregate results, stemming clearly improves retrieval effectiveness for AOC, but not necessarily for Rhino. Given the median number of words in each query (2 for AOC, 4 for Rhino), it is possible that stemming plays more of a role when there are few query terms. We also hypothesize that stemming plays less of a role in Rhino's feature-based concerns than for the action-oriented concerns in the AOC set because of the importance verbs play in the queries. The documentation-based Rhino concern set is similar to the problem of documentation to source code traceability link recovery, where using nouns alone has been demonstrated to improve retrieval accuracy [35]. In contrast, the action-oriented concerns have a strong reliance on verbs in retrieving relevant methods. Since verbs have many forms that lend themselves to stemming, stemming may play a greater role in searching for concerns where an action, or verb, in the query has an impact on retrieval effectiveness.

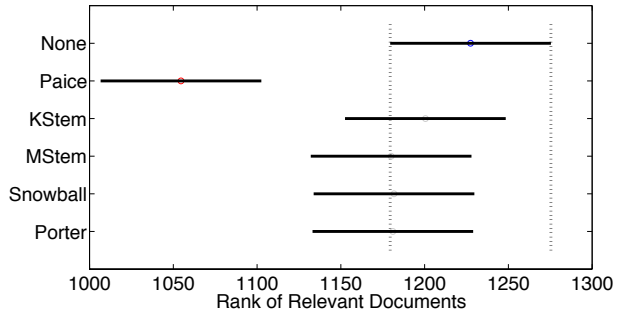
Given the Q_{vary} column in Table X, we see that whether or not stemming improves performance depends on the particular stemmer for more than 35% of the queries. Hence, we further investigated the relative effectiveness of the stemmers on the Q_{vary} set. Table V shows the relative ordering of stemmer performance on the Q_{vary} set, using the Rank Measure F-tests described in Section III-A. Figures 2(a) and 2(b) show the mean and variance rank measure for each type of query in the Q_{vary} set. The x-axis denotes the ranks of the relevant documents and each horizontal line indicates the variance, centered about the mean. Figure 2(a) shows that for AOC, Paice and MStem rank relevant methods more than 100 ranks higher on average than KStem, Snowball, and Porter. For Rhino, Figure 2(b) illustrates that Paice significantly outperforms the other stemmers for this subset.

To further investigate the effect of stemming on concern location, we manually inspected the results for any queries that lead to different results between stemmers.

1) AOC: Of the 29 unique queries in the AOC set, 26 exhibited a difference between stemmers or no stemming.



(a) AOC Concerns



(b) Rhino Concerns

Figure 2. Distribution of Ranks for $q \in Q_{vary}$ for Concern Location

We categorize our observations based on when stemming helps, sometimes hurt, or when the most effective stemmer depends on the particular query.

Stemming > No Stemming (Q_+): As expected, we found some concerns where using any stemming outperformed using no stemming. For the 'compile report' concern using the 'compile' query, conflating 'compile' with related terms 'compiler' and 'compiling' gives stemmers the edge over not stemming. Although no stemming is slightly more effective in the top 10 results for the query 'IReportCompiler', only stemmers are able to achieve 100% recall.

For the 'view listing' and 'sniping' queries, stemming was required to find any relevant results. For 'view listing', all the stemmers in the study conflated 'list' with the query word 'listing'. Although the relevant methods do not include 'listing', they do store the listings in a 'list', thus necessitating stemming to return any relevant methods. Stemming is also required to find relevant results for the 'sniping' query, because the relevant results contain the related term 'snipe', rather than 'sniping'.

No Stemming > Some Stemming (Q_{vary}): As has been found in the information retrieval community [11], we have confirmed that stemming can reduce effectiveness under certain conditions. For example, for queries 'search file' and 'snipe', not stemming returns relevant results at higher thresholds than any other stemming technique used in the study. For 'search file', conflating the plural 'files' causes irrelevant results to be ranked more highly than relevant results when stemming is employed. Similarly for the 'snipe' query, conflating the related word 'sniping' slightly reduces effectiveness. However, the 'snipe' and 'sniping' results illustrate how stemming can improve search consistency—although stemming does not yield the best results for 'snipe', it does yield the same results regardless of whether the query is 'snipe' or 'sniping'. These consistent results outperform no stemming for 'sniping', which returns no relevant results.

The 'add auction' query shows another example where not stemming can be more effective than using stemmers such as Porter or Snowball. Given the overall MAP for each stemmer, we find that KStem, MStem, and Paice > No Stemming > Porter and Snowball. In this case, the morphological and greedy stemmers have an advantage over not stemming or using light rule-based stemmers such as Porter or Snowball, since Porter and Snowball do a poor job conflating 'adding' and 'added' with 'adds' and 'add'.

Threshold-dependent (Q_{vary}): Queries 'play', 'play file', and 'play track', demonstrate the impact threshold selection has on maximizing effectiveness. Within the top 100 results for 'play' and 'play file', at least three different groups of stemmers take the lead. For 'play', MStem and Paice greedily find more relevant methods in the top 20 results because they both conflate 'play' with 'player', unlike KStem, Porter, and Snowball. MStem and Paice reach 92% recall within 120 results returned, whereas KStem, Porter, and Snowball only reach 75% recall.

Paice > other stemmers (Q_{vary}): Given Paice's tendency to overstem and its poor performance in bug localization in the prior section, Paice performed surprisingly well when searching for concerns. Upon further investigation, Paice's overstemming is counteracted by the tf-idf search mechanism. For example, consider the "insert textfield" concern, where 'textfield' is the most effective query, and adding query words like 'report' or 'element' degrades search performance. For the query 'textfield report element', both Paice and KStem are more effective. KStem is more effective because it does not conflate 'element' with 'elements' like MStem, Porter, and Snowball. In contrast, Paice is more effective because it *overstems*, causing the idf term in the tf-idf equation to be so low that its performance is almost the same as the query 'textfield report'. The reason the idf is so low is because Paice incorrectly conflates 'element' with 'el', 'elements', and most importantly, 'else'. Because 'else' is so prevalent in source code, the idf of 'element' reduces its impact on the tf-idf score. In contrast, Paice outperforms the other stemmers for queries 'save auction' and 'save auctions' by *understemming*. In this example, Paice fails to conflate 'saved' with 'save', and ranks two irrelevant results lower.

2) Rhino: Out of 645 queries for the documentation-based concern set, stemming improved or made no change in effectiveness for 182 queries (28%), hurt effectiveness for 239 queries (37%), and had mixed performance for 224 queries (35%). Similar examples for the AOC set can also

be seen for the Rhino set. Rather than repeat examples for Q_+, $Q_=$ and Q_-, we focus on the examples that reveal the widest differences between the stemmers in the Q_{vary} set.

Similar to the AOC set, Paice tends to perform well overall. For example, Paice does not conflate 'set' with any other terms. In contrast, all the other stemmers conflate additional terms with 'set', such as 'sets', 'setting', and 'settable', causing irrelevant results to be ranked more highly than the ones containing 'set'. In addition, conflating other terms with 'set' increases the overall document frequency and decreases the idf, causing documents containing 'set' to be ranked lower than for no stemming or Paice. In another example, Paice's greediness with respect to the word 'operator' increases effectiveness over all other techniques, because highly relevant documents contain the abbreviation 'op', which Paice alone conflates with 'operator'.

Despite Paice's overall effectiveness, there are some examples where Paice's greediness hurts performance. For example, consider the "atan" concern. Unlike the other stemmers and no stemming, Paice conflates the relevant term 'atan' with irrelevant terms 'at' and 'attr'. Thus, Paice finds just one relevant document in the top 20 results, whereas all the other techniques find 19 relevant in the same range.

Although MStem performs well across all ranks, it does not perform as well at the higher ranks. For example, MStem has reduced performance on the "substring" concern compared to other stemmers by not concatenating the abbreviations 'substr' and 'substrs' with 'substring'. In another example, conflating 'small' with 'smallest' causes MStem's document frequency for 'small' to change by one, moving a block of 40 irrelevant results above a block of 8 relevant ones.

VI. Discussion and Conclusions

In this paper, we investigated the use of stemmers on the domain of software, specifically, Java source code. The two types of search tasks—concern location and bug localization—provide us with queries that are complementary in nature. To investigate the effectiveness of the stemmers, we used an evaluation scheme that narrows down the analysis to a set of queries that show high-variance (with respect to stemmer choice) in the rankings of the relevant documents. Further investigation of the queries from this set reveals relative weaknesses and strengths of the stemmers.

Our experiments with bug localization illustrate that there is a clear relationship between the nature of the query and retrieval performance. Whereas the short iBugs queries are closer to natural language text, the longer queries contain code snippets. Intuitively and as confirmed by our experiments, stemming has relatively little effect on the latter type of queries. The short queries displayed the highest variability in the rank measure with respect to the choice of the stemmer used. This variability makes it difficult to recommend any single stemmer when the queries are short (and similar to natural language queries). On the other hand, long queries typically contain code snippets and therefore may not require any stemming in order to lead to better retrievals. A light stemmer like KStem may therefore be desirable for such queries. Across all the three categories—short, medium, and long—the MStem stemmer performs the best mainly because it combines the power of KStem and Paice and is sensitive to conflation of identifiers unique to the software context.

In our concern location experiments, we observe more improvement in retrieval effectiveness for short, verb-heavy queries (AOC) than for noun-heavy, documentation-based queries that are twice as long (Rhino). This result provides further support that the nature of the query considerably affects the success of stemming in improving retrieval effectiveness. Across both sets of concerns, the heavier stemmers (Paice, KStem, MStem) tend to outperform the lighter rule-based stemmers (Porter, Snowball). When the focus is on the most highly ranked documents, Paice and KStem appear the best for concern location, but when ranks of all relevant documents are considered equally, Paice and MStem tend to outperform KStem.

It is interesting to note that while Paice performs well for concern location, it has poor performance for bug localization. We hypothesize that for concern location Paice's greediness is offset by the tf-idf filter that reduces the impact of words that occur throughout the source code. When Paice is at its greediest and least accurate, idf reduces the impact of those words. In contrast, the collection model in the Unigram representation tends to give higher importance to the most commonly occurring terms. Thus, for the Unigram model, Paice's greediness adds noise to the document as well as to the query whenever there is incorrect conflation. Obviously, the extent to which a stemmer is effective depends on the rest of a retrieval framework, including the choice of retrieval model. In future work we plan to investigate the impact of retrieval model on stemmer effectiveness. Another avenue for future work we plan to explore is selectively applying stemming to difficult queries [36].

References

[1] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, & G. Antoniol, "Analyzing the evolution of the source code vocabulary," in *European Conf. on Soft. Maint. & Reeng.*, 2009.

[2] M. Eaddy, A. V. Aho, G. Antoniol, & Y.-G. Gueheneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, & Prog. analysis," in *IEEE Int'l Conf. on Prog. Comp.*, 2008.

[3] G. Gay, S. Haiduc, A. Marcus, & T. Menzies, "On the use of relevance feedback in IR-based concept location," in *IEEE Int'l Conf. on Soft. Maint.*, 2009.

[4] S. Haiduc & A. Marcus, "On the use of domain terms in source code," *IEEE Int'l Conf. on Prog. Comp.*, 2008.

[5] E. Hill, L. Pollock, & K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *IEEE Int'l Conf. on Auto. Soft. Eng.*, 2011.

[6] ——, "Exploring the neighborhood with Dora to expedite Soft. Maint.," in *IEEE Int'l Conf. on Auto. Soft. Eng.*, 2007.

[7] S. Lukins, N. Kraft, & L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Work. Conf. on Reverse Eng.*, 2008.

[8] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, & K. Vijay-Shanker, "Using natural language Prog. analysis to locate & understand action-oriented concerns," in *Int'l Conf. on Aspect-Oriented Soft. Devel.*, 2007.

[9] C. Fautsch & J. Savoy, "Algorithmic stemmers or morphological analysis? an evaluation," *J. of the American Society for Information Science & Technology*, vol. 60, no. 8, pp. 1616–1624, 2009.

[10] W. B. Frakes & C. J. Fox, "Strength & similarity of affix removal stemming algorithms," *SIGIR Forum*, vol. 37, no. 1, pp. 26–30, 2003.

[11] D. Harman, "How effective is suffixing?" *J. of the American Society for Information Science*, vol. 42, no. 1, pp. 7–15, 1991.

[12] D. A. Hull, "Stemming algorithms: a case study for detailed evaluation," *J. of the American Society for Information Science*, vol. 47, no. 1, pp. 70–84, 1996.

[13] R. Krovetz, "Viewing morphology as an inference process," in *Int'l ACM SIGIR Conf. on Research & Devel. in Information Retrieval*, pp. 191–202, 1993.

[14] M. Lennon, D. S. Peirce, B. D. Tarry, & P. Willett, "An evaluation of some conflation algorithms for information retrieval," *J. of Information Science*, vol. 3, no. 4, pp. 177–183, 1981.

[15] C. D. Paice, "An evaluation method for stemming algorithms," in *Int'l ACM SIGIR Conf. on Research & Devel. in Information Retrieval*, pp. 42–50, 1994.

[16] A. Wiese, V. Ho, & E. Hill, "A comparison of stemmers on source code identifiers for Soft. search," in *IEEE Int'l Conf. on Soft. Maint. & Reeng.*, 2011.

[17] M. Porter, "An algorithm for suffix stripping," *Prog.*, vol. 14, no. 3, pp. 130–137, 1980.

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

[19] J. Lovins, "Devel. of a stemming algorithm," *Mechanical Translation & Computational Linguistics*, vol. 11, pp. 22–31, 1968.

[20] C. D. Paice, "Another stemmer," *SIGIR Forum*, vol. 24, pp. 56–61, 1990.

[21] D. Hull, "Using Statistical Testing in the Evaluation of Retrieval Experiments," in *International ACM SIGIR Conference on Research & Development in Information Retrieval*, 1993, pp. 329–338.

[22] M. D. Smucker, J. Allan, & B. Carterette, "A comparison of statistical significance tests for information retrieval evaluation," in *ACM Conf. on Information & Knowledge Mgmt.*, 2007.

[23] B. A. Carterette, "Multiple testing in statistical analysis of systems-based information retrieval experiments," *ACM Trans. Inf. Syst.*, vol. 30, no. 1, 2012.

[24] D. Lawrie, H. Feild, & D. Binkley, "An empirical study of rules for well-formed identifiers," *J. of Soft. Maint. and Evolution*, vol. 19, no. 4, pp. 205–229, 2007.

[25] E. Enslen, E. Hill, L. Pollock, & K. Vijay-Shanker, "Mining source code to automatically split identifiers for Soft. analysis," *Int'l Work. Conf. on Mining Soft. Repositories*, 2009.

[26] A. Marcus & J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Int'l Conf. on Soft. Eng.*, 2003.

[27] V. Dallmeier & T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," in *IEEE/ACM Int'l Conf. on Auto. Soft. Eng.*, 2007.

[28] S. Rao & A. Kak, "Retrieval from Soft. libraries for bug localization: a comparative study of generic & composite text models," in *Work. Conf. on Mining Soft. Repositories*, 2011.

[29] J. Lafferty & C. Zhai, "A Study of Smoothing Methods for Language Models Applied to information retrieval," *ACM Trans. Information Systems*, pp. 179–214, 2004.

[30] M. P. Robillard & G. C. Murphy, "Representing concerns in source code," *ACM Trans. on Soft. Eng. & Methodology*, vol. 16, no. 1, p. 3, 2007.

[31] D. Poshyvanyk, M. Gethers, & A. Marcus, "Concept location using formal concept analysis & information retrieval," *ACM Trans. on Soft. Eng. & Methodology*, vol. 21, no. 4, 2012.

[32] C. D. Manning, P. Raghavan, & H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[33] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, & D. Liu, "Source code exploration with Google," in *IEEE Int'l Conf. on Soft. Maint.*, 2006.

[34] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, & A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Trans. on Soft. Eng.*, vol. 34, no. 4, pp. 497–515, 2008.

[35] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, & S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *IEEE Int'l Conf. on Prog. Comp.*, 2009.

[36] S. Haiduc & A. Marcus, "On the effect of the query in IR-based concept location," in *IEEE Int'l Conf. on Prog. Comp.*, 2011.