A Case Study on the Impact of Similarity Measure on Information Retrieval based Software Engineering Tasks

Md Masudur Rahman · Saikat Chakraborty · Gail Kaiser · Baishakhi Ray

Abstract Information Retrieval (IR) plays a pivotal role in diverse Software Engineering (SE) tasks, e.g., bug localization and triaging, code retrieval, requirements analysis, etc. The choice of similarity measure is the core component of an IR technique. The performance of any IR method critically depends on selecting an appropriate similarity measure for the given application domain. Since different SE tasks operate on different document types like bug reports, software descriptions, source code, etc. that often contain non-standard domain-specific vocabulary, it is essential to understand which similarity measures work best for different SE documents.

This paper presents two case studies on the effect of different similarity measure on various SE documents w.r.t. two tasks: (i) *project recommendation*: finding similar GitHub projects and (ii) *bug localization*: retrieving buggy source file(s) correspond to a bug report. These tasks contain a diverse combination of textual (i.e. description, readme) and code (i.e. source code, API, import package) artifacts. We observe that the performance of IR models varies when applied to different artifact types. We find that, in general, the context-aware models achieve better performance on textual artifacts. In contrast, simple keyword-based bag-of-words models perform better on code artifacts.

Md Masudur Rahman
University of Virginia
Tel: +14344667304
E-mail: masud@virginia.edu

Saikat Chakraborty
University of Virginia
E-mail: saikatc@virginia.edu

Gail Kaiser
Columbia University
E-mail: kaiser@cs.columbia.edu

Baishakhi Ray
Columbia University
E-mail: rayb@cs.columbia.edu

On the other hand, the probabilistic ranking model BM25 performs better on a mixture of text and code artifacts.

We further investigate how such an informed choice of similarity measure impacts the performance of SE tools. In particular, we analyze two previously proposed tools for project recommendation and bug localization tasks, which leverage diverse software artifacts, and observe that an informed choice of similarity measure indeed leads to improved performance of the existing SE tools.

1 Introduction

Information retrieval (IR) techniques play a pivotal role in many Software Engineering (SE) tasks. Haiduc et al. identified more than 20 different SE tasks [1], e.g., feature location, traceability link recovery, bug localization and triaging etc., that use IR or Text Retrieval (TR) techniques[1]. Typically, an IR method depends on two key components: (i) *query*: contains users' information needs, and (ii) *candidate document*: from which the relevant information is extracted. Given a query, an IR task finds relevant candidate documents by computing a similarity score between the query and each document using an appropriate similarity measure. The candidate documents are then ranked based on the decreasing values of the similarity scores. Thus, measuring document similarity is one of the key components of any IR technique. Exact matching (e.g., Boolean), matching based on similar bag-of-words (e.g., VSM [3]), context-based matching (e.g., LSI [4], WMD [5]), etc. are a few popular methods for measuring document similarities.

Usually, well-established IR techniques are used to measure document similarities for SE tasks, primarily because they are already stable, fine-tuned, and well-explored. However, these models are refined mainly for natural language (NL) text corpora. SE corpora, which usually contain a diverse set of information including source code, bug reports, project descriptions, API documentations, etc., are linguistically different from natural language [6] due to the following properties: (i) *less ambiguous*: Documents containing source code are less ambiguous than NL so that compiler can interpret the code. (ii) *open vocabulary*: Source code contains open vocabulary where developers can coin new variable names without changing the semantics of the programs. (iii) *diverse*: Different types of SE documents can have different language properties. For example, a bug report that primarily contains NL text with domain-specific keywords is linguistically very different from source code or execution traces. Thus, while applying the IR models to SE context, we have to adapt the models to cater these SE characteristics—a best performing similarity measure, which is tuned for NL tasks, may not be optimal for SE corpora. In fact, different similarity measures might be suitable for different type of SE artifacts.

In this paper, we extensively evaluate how the performances of SE tasks vary with the choices of different similarity measures. In particular, we present

---

[1] TR is a branch of IR [2]. We use the terms interchangeably.

two case studies: (i) *project recommendation*: given a GitHub project as query, it tries to find functionally similar GitHub projects, and (ii) *bug localization*: given a bug-report as query, it retrieves relevant buggy source file(s). We carefully choose these tasks as representative SE jobs because they involve contrasting diverse sets of SE artifacts. In particular, for project recommendation, the similarity is measured among different textual (e.g., text vs. text) and code (e.g., code vs. code) artifacts separately, whereas the latter relies on the similarity computation between mixtures of textual and code document types (e.g., code vs. text).

We evaluate the effectiveness of four popular IR measures: Vector Space Model (VSM) [3], Latent Semantic Indexing (LSI) [4], BM25 [7], and embedding based Word Mover's Distance (WMD) [5] on different SE documents. Each of these measure has its own benefit and therefore has its preferred domain. For example, in SE literature, VSM [8–12] and LSI [13–15] are commonly used, while BM25 is popular in general purpose search engines, and WMD is the state-of-the-art similarity measure for natural language document classification. Recently, researchers start proposing word embedding based models to improve different SE tasks [16–18]. Thus, we choose different measures that are known to be effective for different corpus and tasks, and analyze them thoroughly for SE artifacts.

We study 1832 GitHub projects for the project recommendation task and 1100 bug reports for the bug localization task. We observe that context-aware models such as LSI and WMD are in general better, while the keyword based bag-of-words (BOW) model VSM performs best for code-only artifacts. In contrast, BM25 performs the best on mixed artifacts. Surprisingly, BM25 is not as effective for text- or code-only documents.

To further investigate the impact of such an informed model choice, we conduct a study on two previously proposed tools: CLAN [13] for project recommendation and LR [8] for the bug localization task. CLAN leverages LSI to compute the similarity between two type of code artifacts: package and API. We replaced the LSI similarity measure with VSM, as we observe that the VSM model is optimal (among the IR models considered) for code-only artifacts. Experiments on our dataset confirmed that the informed model choice indeed improves the performance of this project recommendation tool. We conducted a similar experiment on the bug localization task. Ye et al.'s bug localization tool LR [8] leverages VSM to compute the similarity between a bug report and source code (i.e. mixture of text and code). Since our experiments show that BM25 is the optimal model choice (among IR models considered) for such mixtures, we then replace the VSM similarity measure with BM25. We observe that our modification indeed improves the performance of the bug localization on a benchmark dataset.

We make following contributions:

1. We evaluate four IR-based similarity measures on diverse artifacts w.r.t. two SE tasks.
2. We provide empirical evidence that an optimal choice of similarity measure can improve the accuracy of these two IR-based SE tasks.

3. We curate a valuable dataset of 1832 GitHub projects by retrieving their descriptions, readme contents, method class names, imported package usage, and API for project recommendation. We manually associate each project with a fine-grained category that describes their functionalities. Our dataset is available open source at `https://drive.google.com/drive/folders/1wqViscmoD_ikFuxxbcF0aTHD3gvnWWw4?usp=sharing`.

The rest of the paper is organized as follows: Section 2 presents relevant technical background. Next, in Section 3, we discuss methodology for project recommendation and bug localization tasks, including data collection, feature extraction, and evaluation metric. We discuss our case study in Section 4 w.r.t. the two SE tasks including research questions, experimental results, and implications. In Section 5, we summarize our experimental findings with possible implications and future works. We discuss related works in Section 6. Finally, in Section 7, we examine potential threats to validity that may affect our findings, and we conclude in Section 8.

2 Background

In this section, we present the relevant technical background.

2.1 Similarity Measure

1. Vector Space Model (VSM). In VSM, documents (D) and queries (q) are represented as N-dimensional vectors where N is the size of the vocabulary, and each dimension corresponds to a separate word or term. Each vector element represents the weight of the corresponding term; i.e., $q = (qw_1, qw_2, qw_3, ..., qw_N)$ and $D = (Dw_1, Dw_2, Dw_3, ..., Dw_N)$ where the qw_i and Dw_i are the weights of the term i in a "bag-of-word" representation of vocabulary size N. An effective way to compute the term weight is the term frequency-inverse document frequency (tf-idf), where tf represents the importance of the term in a document, and idf represents how valuable or rare the term is across all the documents. Then the similarity between two documents is computed as the cosine angle between corresponding vectors as $sim(q, D) = \cos(q, D) = \frac{q^T D}{||q|| \, ||D||}$. The high cosine value means the two documents are similar.

Implication: VSM model is effective and simple to implement. However, since it is a BOW and exact keyword matching approach which ignores the order of the tokens in a document, this method is suitable when the order of the words does not matter. Note that in the cosine similarity formula, the magnitudes of the document vectors ($||q||$ and $||D||$) are in the denominator and give smaller cosine value for the larger dimensional vector. Thus, longer documents may be penalized because they have more components that are indeed relevant.

2. BM25. Also known as Okapi BM25 (BM stands for Best Matching), this BOW based probabilistic retrieval model ranks documents based on the number of query terms present in each document. BM25 ignores inter-relationship between the query terms. For a given query q with query terms $q_1, q_2, q_3, ..., q_n$, the BM25 relevance score of document D can be calculated as:

$$score(q, D) = \sum_{i=1}^{n} IDF(q_i)\left(\frac{tf_i(k_1+1)}{tf_i+k_1(1-b+b\frac{|D|}{avg|D|})}\right)\left(\frac{qtf_i(k_2+1)}{k_2+qtf_i}\right).$$

where, for the query term q_i, IDF is the inverse document frequency, and tf_i and qtf_i are term frequencies w.r.t. to D and q respectively. $|D|$ is the document length and $avg|D|$ is the average document length across all the candidate documents in the corpus. Here, b, k_1, and k_2 are hyper parameters. Notice that, k_2 controls the query term frequency and can be ignored for short query like free text search where frequency per query term is usually 1. However, in our settings, depending on the SE artifacts, the query can be long and may contain repeated terms that may have important contributions towards the overall similarity measurement. Hence, we keep this k_2 hyper parameter in the equation.

Implication: A distinguishing feature of BM25 is that it treats a matching term's importance in the document and query differently and also gives a special attention to that term's frequency in the query. This characteristic helps BM25 to show improved performance when the query and document are of different types. On the other hand, the document length normalization helps to predict more accurate ranked score where the documents are of various length. Despite these advantages, BM25 is also a keyword matching model and ignores orders of the word in a sentence. Thus BM25 might fit well where document length varies and the order of the tokens in the document is not important.

3. Latent Semantic Indexing (LSI). Landauer and Dumais [4] proposed LSI that is based on the assumption that words with similar meaning will have similar context. LSI projects a higher dimensional document-term co-occurrence frequency matrix into a lower dimensional latent space to create document vectors. An effective way of using LSI is to use the tf-idf weight instead of raw co-occurrence count of term. The idf can be estimated from the document corpus. After inferring the lower dimensional vector of both query and candidate documents, cosine similarity can be used to compute the similarity between two document vectors as $sim(q, D)$ (equation above).

Implication: Intuitively, the dimension reduction step computes similarity scores of every word w.r.t. every other based on their co-existence in a common context. In this way, LSI captures the meaning of synonymous and polysemy words in the latent space. As opposed to VSM and BM25, LSI can better differentiate the documents with synonymous and polysemic words but little semantic similarities.

4. Word Embedding. Similar words should have similar context [19]; this observation leads to *Word Embedding*, a natural language processing (NLP) technique, where each word w is represented by a d-dimensional vector of real numbers. This d-dimensional vector is learned from the context of w where *context* is formed by the preceding and following words of w in a sentence. Sim-

ilar words should have similar context words thus similar embedding. Many popular similarity measures like cosine similarity can be used to measure similarities between the embedded documents. Among them *Word Mover's Distance (WMD)* is proved to be the winner [5]. For each query term, WMD searches for the semantically closest document term in each document, where the distance between two terms is calculated as a Euclidian distance in word embedding space. The summation of the minimum distances for all query terms represents the distance from a query to a candidate document.

Implication: As word embedding capture the contextual information of words, WMD is useful to bridge the semantic gap between documents. For example, say the descriptions of two projects are "image gallery app for Lollipop" and "Android photo viewer". They are very close in meaning but have no shared words. Thus, traditional similarity measures like keyword based BOW model could not find any similarity between these two documents. In contrast, WMD can efficiently judge they are highly similar since they have very similar word embedding. In different SE artifacts, some synonymous terminology is common; upgrade and update often used interchangeably. WMD might be useful to detect similar documents even if the documents contain no identical words.

2.2 Studying SE Tasks

We analyze the effect of different similarity measure on different types of software document w.r.t. two IR-based SE tasks:

(i) **Project Recommendation.** Given a project as a query, this task tries to find functionally similar projects from GitHub. A ranked list of projects is retrieved with the most relevant projects at the top. For example, screenbird [20] and FFmpegRecorder [21] both are Video Recorder software. For a query with the first project, the system tries to return a list of Video Recorder projects that includes the second project (see Table 4).

(ii) **Bug Localization.** Given a bug report as query, this task ranks all the source files in the project repository based on their relevance with the query. The files that top the ranking are more likely to contain the cause of the bug. For example, for bug report id 369884 [22] in Eclipse-Platform-UI [23] project, file *E4Application.java* [24] was fixed (see Table 7). A perfect bug localization tool will rank this file at top if queried with above bug report.

3 Methodology

Here, we describe the dataset and the analysis methods we used to conduct the study.

3.1 Study Subject

We analyze a wide variety of projects for studying two SE tasks that are introduced in Section 2.2. For project recommendation task, we use a total 1832 GitHub Java projects of 112 functional categories. Details of the dataset can be found in Table 1a. For the bug localization task, we collect a benchmark bug report dataset [8,16], which contains 1100 bug reports from four projects. Details of the dataset are shown in Table 1b.

3.2 Data Collection

For studying project recommendation, we use GitHub open source projects.
Collecting GitHub Projects. We use two different methods to select GitHub projects. First, in (i) *Method-A*, we search GitHub with keywords representing project functionalities (e.g., media player, text editor, etc.), and download the projects retrieved by the GitHub search. In this way we download 1590 projects with 78 different functionalities. As GitHub search primarily looks at project descriptions, a project without proper description may have suffered from this step. Hence, in (ii) *Method-B*, we download some GitHub projects first and then determine their functionalities. In the following section, we describe these methods in details.

Table 1: **Study Subject**

(a) Project Recommendation task

	#Project	#Category	#Java File	#Method Class	#API	#Import Package
Method-A	1590	78	216K	4.9M	1.5M	2.04M
Method-B	242	55	14K	0.3M	0.1M	0.12M
Total	1832	112	230K	5.2M	1.6M	2.16M

(b) Bug Localization task

Project	Time Range (mm/yy)	# bug reports	# Java files in versions median	total	# API entries
Birt	06/05 -12/13	200	8770	1770K	957
Eclipse-UI	10/01 - 01/14	200	6141	1228K	1314
JDT	10/01 - 01/14	500	8819	4421K	1329
SWT	02/02 - 01/14	200	2794	559K	161

Method-A. Given a project functionality, we use GitHub API to search for the corresponding projects—project functionalities are represented by the search keywords. For example, for retrieving projects related to *Video Recorder* application, we search with keyword *Video Recorder*.

We begin with selecting a meaningful set of project functionalities that we can use as keywords to search projects. We leverage DMOZ Ontology [25],

which is a hierarchical directory of the Web. In this ontology, any category under 'software' represents a meaningful functionality (e.g., Spelling Software, Grammar and Spell Checkers, etc.). We remove the homonyms to reduce confusion of the search task. This approach reduces the category set to a size of 90, where each phrase represents a certain software functionality, such as Spell Checker.

We use these categories as queries to search GitHub for relevant Java projects using GitHub search API [26]. We exclude the forked projects as they include near-identical projects and overfit our project similarity data. For each query, we select top 1,000 projects from the search result. Further, we only select projects with ratings 3-star and above to focus on important and (hopefully) well documented projects. Thus, we end up having 2180 unique projects under 90 project categories, where some projects may belong to multiple categories.

We further manually investigate the associated categories of each project, because GitHub search is mostly based on keyword matching and in some cases it leads to inaccurate categorization. For example, projects *Eid-Applet* [27]'s description is "eID Applet to enable BE eID cards within web browsers and it is retrieved by the query Web Browser". The retrieved project is certainly not a Web Browser but an Applet. While manually investigating the project annotation, we further modify, delete, and add categories (i.e. functionalities) as needed. We also remove some ambiguous projects. After such rigorous filtration process, we end up with 1590 projects under 78 different functionalities. Table 1a presents the details.

Notice that, the collected projects are popular (above 3-star rating) and resulted from a keyword based search. Thus, most of the projects contain proper description, readme, and other source code content. However, there are many projects on GitHub that do not have adequate textual artifacts (description, readme, etc.). To study well-represented diverse projects, we further enhance the data set using the following method.

Method-B. First, we collect 242 GitHub projects that have Google play links in their descriptions or README contents. Note that we exclude forked projects and consider popular project having at least 3-start to remove any potential toy project [28]. Then we manually annotate them using the details available in the Google Store. We leverage the app description, similar app suggestion, category, etc. information available in Google Play Store to annotate these projects. In this way, we can annotate a GitHub project with its functionality even if it does not contain elaborate textual artifacts except Google play link (in description or readme). We also find some new project functionalities that have not seen in Method-A. Finally, we find 1832 (1590 + 242) projects with 112 different functionalities (see Table 1a), where Media Player, Search Engine, Database Systems, etc. are top functionalities with maximum member projects.

For both methods, two researchers annotated project category *separately* and resolved the disagreements by discussion. We observe that 95% of the cases they agreed on project categories. Note that, as the annotator needs to consult various documents (i.e. description, readme, Google store, etc) to come

up with a functional category, it required tremendous manual effort to annotate all the projects. Per annotator, it took on average 3 minutes per project and in total approximately 90 working hours to finish all the annotations. We will open-source our dataset to encourage further research.

Collecting Bug Report Data. We collect a benchmark bug report dataset and study four projects' data: Birt [29], Eclipse Platform UI (Eclipse-UI) [23], Eclipse JDT [30], and SWT [31]. This dataset has been used previously for the bug localization task [16,8,9,32]. In this dataset, each bug report contains a summary, description, report time, and status of its fix along with bug fix commit. For each bug report, using the bugfix commit, we download the before-fix version of the project. The files that are added, deleted, and modified in the bugfix commit are considered as the true buggy files. As the added files are not part of the before-fix version, the system cannot predict it, so we ignore that for the evaluation. Table 1b shows details of the report dataset.

3.3 SE Artifacts Extraction

We extract following documents from our dataset w.r.t. two SE tasks.

Features of project recommendation task. From the collected project data, we choose five types of SE artifacts:

(i) Project Description: This textual artifact is often short and concisely represent the project functionality.

(ii) Readme Content: Textual artifact, usually contains a detailed description including how to install and run the project.

(iii) Method & Class name: Developers often use meaningful identifier names to implement project [33]. Thus, it might be possible that projects with similar functionalities use similar method or class names. For example, two text editor applications may have similar methods with names copy, paste, save etc. To check this hypothesis, we retrieve method and class names that are declared within a project.

(iv) Import Package name: Similar projects often use similar API packages [13]. This motivates us to use imported API package names and class names as features. We use Eclipse JDT [30] framework to collect these names.

(v) API name: The API Class refers to the classes defined in system libraries or other third-party libraries or packages. To extract these, using Eclipse JDT we first extract all the classes used in a project and then remove the classes defined within the project from this list. The remaining class names are considered as API names.

Features of bug localization task. For each project we consider the source files, which mostly contain code tokens, as candidate documents to compute their similarity with the bug report query.

3.4 Data Pre-processing

For each feature, we use standard natural language processing (NLP) techniques for data processing like tokenization, normalization, stemming, and

stopword removal. First, we clean the documents by removing the special character (i.e. non-English) and punctuation from it. As a convention, Java usages camel case format for its class, method or any variable name. For such compound tokens (i.e. *TerminalFactory*), both in textual and code artifacts, we further extract smaller token units (i.e. *Terminal* and *Factory*). We also keep the original compound token in code artifacts to keep actual keyword information. We then normalize the tokens: remove the numeric character and convert to lower case letter. To avoid bias from the frequently occurring but less informative tokens we remove two types of stopword: standard English stopword list (adopted from [34]) and Java language related stopwords, that is keywords [35]: *void*, *public*, *while*, etc. To reduce the unwanted lexical gap between tokens, we apply Porter Stemmer[36] to convert words to its base form (i.e. convert *computes* and *computed* into *comput*).

3.5 Evaluation Metric

We evaluate an IR task w.r.t. its ground truth sets, i.e., given a query and a candidate document, we check whether the retrieved results are matched with its corresponding ground truth. We use several standard evaluation metrics [37] as described below:

1. Precision (P). For a given query q, precision is the fraction of retrieved documents that are also present in the ground truth set. Thus, $P = \frac{r}{d}$, where r is the number of relevant items from the retrieved d documents.

2. Recall (R). For a given query q, recall is the fraction of relevant documents that are retrieved. If t be the total relevant documents for the query q, the recall is $R = \frac{r}{t}$.

3. Mean Average Precision (MAP). For a set of queries, MAP is the mean of the average precision of individual query [37]. First, for each query, an average precision is computed for each rank. Given a query(q) and it's ranking documents, average precision of q is calculated as $AvgPrec(q) = (\sum_{i=1}^{R} \frac{i}{rank_i})/R$, where R is the total number of relevant documents, $rank_i$ is a ranking position of the relevant document i in the retrieved ranking and $i/rank_i = 0$ if relevant document i was not retrieved by the model. Then we take the mean of this average precision across all the queries using equation $MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} AvgPrec(q_j)$ to get MAP. Here, Q is the entire query set.

4. Mean Reciprocal Rank (MRR). Given a retrieved list for a query, the reciprocal rank is computed as the multiplicative inverse of the rank of the first relevant document. Mean of such reciprocal rank across all the queries are taken using equation $MRR(Q) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$. Here, $rank_i$ is the rank position of the first relevant document for the ith query.

We evaluate a search result by computing these evaluation metrics at different rank cut-off. During comparison we use percentage gain computed as $gain = (b - a)/a * 100$, any metric value changes from a to b.

3.6 Model Configurations

Performance of IR models varies significantly with different parameter settings [38,39]. For a fair comparison, we tune all the models to its best performing configuration for each task, as shown in Table 2. We did an exhaustive search—we varied the parameter values at regular intervals and then chose the optimal ones.

Table 2: **Best performing models' configurations**

	VSM	**BM25**				**LSI**		**Word2Vec**			
	Min DF	Min DF	k1	k2	b	Min DF	Projected Dim.	Min DF	Dim.	Window Size	Vocab Size
project recommendation	2	2	1.5	1.5	0.75	2	100	5	300	5	18M
bug localization	1	2	1.5	1.5	0.75	15	100	5	100	10	21.8K

We train a skip-gram based word2vec [40] word embedding model which is used by WMD. We use a diverse collection of $3.7M$ Wikipedia articles [41] and $7.5M$ GitHub projects description collected using GHTorrent [42] for the training data. We use Gensim's [43] Python implementation of word2vec to train on our data. For WMD in bug localization task, we use a pre-trained word2vec model which is trained on the data containing source code and API documentation and found to be effective on the same dataset for bug localization [16].

4 Case Studies

Our central question is whether a choice of similarity measure matters while computing similarities between different types of SE artifacts, especially for IR related SE tasks. Thanks to the software forges like GitHub, Bitbucket, Bugzilla etc. various types of textual and code related SE artifacts are available. First, we collect a ground truth set of closely matched documents corresponding to the two studied tasks. Next, we check how different similarity measures perform to find relevant documents w.r.t. to this ground truth set. Inspired by our empirical results, we investigate whether such an informed choice of similarity measure impact the performance of IR based SE tools. We analyze two previously proposed tools for project recommendation and bug localization tasks.

4.1 Case 1: Project Recommendation

In project recommendation task, for a given project set N, we take one project as the query and consider rest $N-1$ projects as candidate documents. If the

Table 3: Different models performance on **textual artifacts** for project recommendation task. Best performing values marked in Red (**bold**).

	Description				Readme			
	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD
MAP@10	0.51	**0.57**	0.51	0.51	0.37	**0.39**	0.26	0.29
MRR	0.56	**0.61**	0.55	0.57	0.44	**0.45**	0.30	0.34
P@10	0.40	**0.49**	0.41	0.33	0.25	**0.29**	0.14	0.14
R@10	0.13	**0.16**	0.13	0.10	0.07	**0.08**	0.03	0.03

categories of a retrieved project and the query project are identical, we consider that as a success. We take the average performance across all the projects in a randomly chosen query set of 200 to get overall performances of different models. We discuss the results mostly w.r.t. $MAP@10$. However, a similar conclusion can be drawn from all other evaluation metrics.

Text-Text Artifact. We study the impact of various similarity measure while computing similarities between textual documents, e.g., description vs. description, and readme vs. readme. Project recommendation task is a classic example of this. Given a project name as a query, the query string is augmented with textual feature documents. The candidate projects are then searched using different similarity measure. We first investigate:

RQ1. How different similarity measures perform for Text-Text artifacts?

In Table 3, we see that for the textual artifacts, in general, LSI achieves the highest performance for most of the evaluation settings. For *project description*, LSI model performs best for all the evaluation metrics and gains 11.76% compared to the second best models (WMD, VSM, and BM25) at $MAP@10$. In our dataset, WMD, VSM, and BM25 perform comparably. Similarly, for *readme*, LSI model performs best in most of the evaluation metrics —4.85% improvement for $MAP@10$ w.r.t. VSM, the second best model.

Table 4 shows example ranked lists for all four models using **description** only feature for a query project 'screenbird' with Video Recorders category. The top two projects retrieved by LSI and VSM models are of same categories as the query. Notice that all these retrieved projects have keyword 'video', and 'recording' in their descriptions. However, VSM mistakenly retrieves a media player app as the third project because of the word 'video' presents in the document description.

Interestingly, all the projects retrieved by WMD in Table 4 are wrong, and they have minimum keyword overlapping with the query projects. However, a closer look will reveal the document and query terms are related. This indicates projects description of GitHub is not very complex. We may not need a word embedding based similarity measure where contextual similarity pays a pivotal role and may undermine simple keyword based matching.

> **Result 1:** *Context-aware model LSI performs the best for text-text documents.*

Table 4: Top three Search Results for project recommendation task with **description** only feature. The keywords in **bold** highlight the important keywords for matching.

Category	Name : Description
Query	
Video Recorder	screenbird :a full **cross platform video screen** capture tool and **host**. Java based **screen recorder**, and Django based **web** backend distributed **video** processing **engine** that uses **ffmpeg** ... of AWS instances.
LSI	
(1) Video Recorder	FFmpegRecorder : An Android **video recorder** using ... **FFmpeg**.
(2) Video Recorder	FFmpegVideoRecorder : Customizable Android **video recorder** library...
(3) Video Recorder	jirecon : A Standalone **recording** container for ... **video recorder** ...
VSM	
(1) Video Recorder	ScreenRecorder : containing service for **recording video** of device screen
(2) Video Recorder	VideoRecorder : Android **video recorder** project
(3) Media Player	dttv-android : android **video** player based on dtplayer
BM25	
(1) Video Recorder	ScreenRecorder : containing service for **recording video** of device screen
(2) Terminal Emulator	DragonConsole : a **cross platform** Java based Terminal Emulator.
(3) Search Engine	LunarBase : real-time engine, ... **records** in one table, ... used as a search **engine**
WMD	
(1) Media Player	supersonic : **web**-based **media streamer** ... **audio** and **video** formats
(2) Readers Java	java-manga-reader : directly from **web**. .., **internet** access is required
(3) Search Engines	SearchEngine : crawls seed **web** page ... search **engine** for a **website** ...

Code-Code Artifact. Similar to textual artifacts, we study the impact of various similarity measures while computing similarities between code documents, e.g., method-class vs. method-class, and API vs. API, etc. We leverage document features from project recommendation for this RQ. Next, we investigate:

RQ2. How different similarity measures perform for Code-Code artifacts?

While the performance of LSI is higher in textual artifacts, VSM model starts dominating in performance for code artifacts. Compare to LSI, VSM performs slightly better at *method and class* feature and significantly better for *import package* and *API* names with 20.59% and 24.29% improvement respectively at $MAP@10$.

Among code features, method and class names perform best. This suggests that similar projects actually have similar method and class names. This finding also confirms the hypothesis of Allamanis et al. [33] that developers use meaningful identifier names while writing a software program.

Table 5: Different models performance on **code artifacts** for project recommendation task. Best performing values marked in Red (**bold**).

	Method Class				Import Package				API			
	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD
MAP@10	**0.37**	0.36	0.16	0.25	**0.29**	0.24	0.07	0.20	**0.31**	0.25	0.22	0.25
MRR	**0.43**	0.40	0.19	0.29	**0.34**	0.29	0.08	0.26	**0.35**	0.28	0.26	0.30
P@10	0.23	**0.24**	0.07	0.10	**0.16**	0.13	0.03	0.08	**0.17**	0.13	0.10	0.12
R@10	0.06	**0.07**	0.02	0.03	**0.05**	0.03	0.01	0.02	**0.05**	0.04	0.04	0.04

Interestingly, both the context aware models LSI and WMD show a similar decreasing trend of performance going from textual artifacts to code artifacts (see Table 5). This result indicates that the contextual feature, which is quite effective in natural language text, is not that helpful for source code artifact. Further, among source code artifacts, LSI and WMD perform better for method class names suggesting context is more meaningful for these names than import packages and API names.

On the other hand, BM25 performs significantly worse for the rest of the artifacts. BM25 treats query and document differently (see the scoring equation in Section 2), although for homogeneous artifacts (e.g., code vs. code and text vs. text) the query and documents are linguistically identical. In addition, BM25 assumes all the terms in the query are important. For example, in Table 4, except for the top project, the rest are not correct for BM25. As it ignores inter-relationship of query terms, it emphasizes on all parts of the query and thus misguided by the variety of concept in the query. Thus, a verbose query might hurt the performance of BM25.

Result 2: *Keyword based bag-of-words model VSM performs best for code only artifacts. Code artifacts lack context information.*

4.1.1 Implication

Previously proposed tool, CLAN [13] compares JDK APIs (packages and classes) used in the studied projects using LSI algorithm to establish similarities. Since CLAN's source code is not available, we reimplement CLAN adhering to the paper details. We further extend CLAN to incorporate all the APIs studied in our dataset including JDK APIs.

We find that VSM is the best performing measure for the artifacts type used by CLAN (see RQ2). Thus, we build a modified version of CLAN, named vsmCLAN, where we replace the similarity model used by CLAN with VSM. CLAN usages a weighted combination of both features' similarity value for the final score. Thus, we exhaustively tune the weights in an interval of 0.1 and report optimal performance for the comparison. For this experiment we consider a query set consisting of 1590 projects (i.e. method-A in Table 1a) and report average performance. We observe that original CLAN achieve the best results at weights *import package* = 0.9 and *API* = 0.1. and modified vsmCLAN achieve the best results at weight *import package* = 0.6 and *API* = 0.4. Next, we analyze this research question.

RQ3. Can an informed choice of IR model improve the performance of project recommendation task?

Figure 1 shows that our modified version vsmCLAN outperforms the original tools (lsi)CLAN by 35% at MAP@10, 35% at MRR, 25% at P@10, and 29% at R@10. Though all the system architecture remains the same for the modified version, better performance of VSM on code artifacts (i.e. package and API) contributed to the overall improved performance of the tool.
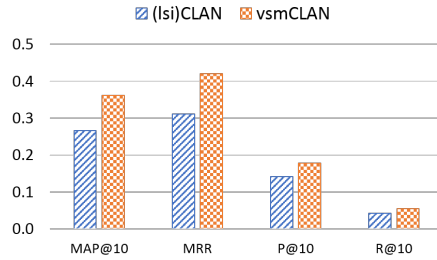

Fig. 1: **Comparison with CLAN**

All these results strongly suggest that with a right choice of similarity measure can really improve the performance of the existing tool.

> **Result 3:** *A recommendation tool built with an informed choice of similarity measure can significantly improve the performance of original tool.*

4.2 Case 2: Bug Localization Task

We empirically evaluate the performance of various similarity measures given the bug reports as a query. Here, we only consider source code as the documents.

Mixture Artifact. In this case, the query and candidate document types are different (text vs. code). We check the similarities between bug reports and the source files by studying 1100 bug reports from four projects (Table 1b). If the model ranks the actual buggy file at the top, we consider that as a success. Next, we question:

RQ4. How different similarity measures perform across a mixture of text and code artifacts?

We observe that BM25 is the best performing model across all the projects, as shown in Table 6. BM25 outperforms other models significantly and achieves a percentage gain of (MAP@10, MRR): Birt (54%, 38%), Eclipse-UI (190%,

Table 6: Different models performance on **mixture artifacts** for bug localization. Best performing models' values marked as Red (**bold**) for each project and evaluation metric.

	Birt				Eclipse-UI				JDT				SWT			
	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD
MAP@10	0.11	0.03	**0.17**	0.02	0.10	0.06	**0.29**	0.02	0.06	0.02	**0.28**	0.00	0.10	0.11	**0.42**	0.01
MRR	0.13	0.04	**0.18**	0.02	0.12	0.08	**0.30**	0.02	0.08	0.03	**0.31**	0.01	0.12	0.13	**0.44**	0.01
P@10	0.03	0.02	**0.05**	0.02	0.04	0.03	**0.08**	0.02	0.02	0.01	**0.07**	0.00	0.04	0.04	**0.10**	0.01
R@10	0.13	0.05	**0.23**	0.02	0.16	0.08	**0.45**	0.02	0.14	0.04	**0.43**	0.00	0.18	0.20	**0.55**	0.01

150%), JDT (366%, 287%), and SWT (320%, 266%), compared to the second best VSM model. Table 7 shows an example, where the BM25 ranks the intended file at top 1 position, where VSM ranks it at 31st places.

Table 7: Sample results for bug localization.

Bug Reports and Fixed File	Rank
Bug 369884 [22] platform:/plugin/ not used for applicationXMI. ... used for CSS resources or Icons. I think the ... the applicationXMI parameter. Also the e4 wizard should be adjusted to create the right URI. **Fixed File : E4Application.java** [24]	**BM25=1** VSM=31 LSI=110 WMD=5983

Here bug report is human written summary of a bug of software thus contains important keyword to locate bug in a software project. Thus, bug reports are usually short in length and free of unnecessary repetition of the term. On the other hand, the source code document contains mostly code token. Though the variation of length of different source file might be responsible for the worst performance of the other models, BM25 mitigate its impact with the document length normalization factor. Thus the nature of query (bug report) and document (source code) makes BM25 a better choice among all the models, which is also reflected by the evaluation results.

VSM and LSI assume both query and document are of same type and represent them in the same concept space. This assumption works better for homogeneous artifacts: text-text in RQ1 and code-code in RQ2. But for mixture artifacts like bug reports vs. source code, representing them in the same concept space is less effective and that is also indicated by the results in Table 6.

Similar to RQ2, we also see that contextual feature in source code is less effective. Due to the the co-existence of text and code in bug reports, the context feature is less in the bug reports as well. This lack of context in the query and document might be a reason for the poor performance of context aware models (LSI and WMD).

> **Result 4:** *For mixture (text-code) documents, BM25 performs the best. Surprisingly, BM25 is not that effective for text only and code onlyartifacts.*

Table 8: Model assignment of features for bug localization task

Feature	Source Code	API Descr.	Collab. Filter	Class Name	Bug Fix Time	Bug Fix Freq
bm25-LR	BM25	BM25	BM25	meta-value	meta-value	meta-value
(vsm)LR	VSM	VSM	VSM	meta-value	meta-value	meta-value

Table 9: Optimal Feature weights for original **(vsm)LR** bug localization tool

Project	Source Code ($w1$)	API Des. ($w2$)	Collab Filter ($w3$)	Class Name ($w4$)	Bug-fix Recency ($w5$)	Bug-fix Freq. ($w6$)
Birt	5	0.5	5.5	5.5	1.5	0.55
Eclipse-UI	9.5	0.95	4.5	6.5	1.05	0.75
JDT	4.5	2.6	5.5	6.5	1.05	0.55
SWT	4.2	3.5	4.7	7.9	0.05	0.95

Table 10: Optimal Feature weights for modified **bm25-LR** bug localization tool

Project	Source Code ($w1$)	API Des. ($w2$)	Collab Filter ($w3$)	Class Name ($w4$)	Bug-fix Recency ($w5$)	Bug-fix Freq. ($w6$)
Birt	2.4	0.05	3.5	2.5	0.6	0
Eclipse-UI	3.4	0.05	3	2.5	0.5	0
JDT	1.2	0.25	2.5	1.2	0.3	0
SWT	4.4	0	3.5	2.5	0.3	0.5

4.2.1 Implication

Ye et al. [8] proposed a learning to rank based bug localization tool (LR) which leverages the VSM similarity measure. We implement LR tool adhering the paper details. LR combines six feature score to get the final score for a source file corresponds to a bug report query. Among six features, three features: Class Name, Bug fix time, bug fix frequency are meta information. We follow the same technique as Ye et al. [8] to compute these feature values for both the tools. Please refer to the Ye et al. [8] for feature extraction details. Ground truth collection from benchmark bug-report dataset can be found in Section 3.

Our experiments show that BM25 performs best for such a mixture of textual and code artifacts. Thus we replace similarity measure of LR tool from VSM to BM25 to build bm25-LR as in Table 8. Note that meta-features do not require any similarity measure, thus both original (vsm)LR tool and modified bm25-LR uses the same value for these features. For this experiment we consider the latest 100 bug reports for each of the four projects (in Table 1b) from our dataset. Note that LR model leverages a learning to rank to find out optimal weights to combine features [8]. Thus the learning weights heavily depend on the training data. To mitigate such unwanted bias, we apply an exhaustive search by varying weights in regular interval (i.e. 0.05) to achieve optimal parameter settings. The optimal feature weights for (vsm)LR and bm25-LR are given in Table 9 and Table 10 respectively. Then, we investigate the following research question.

RQ5. Can an informed choice of IR model improve the performance of bug localization task?
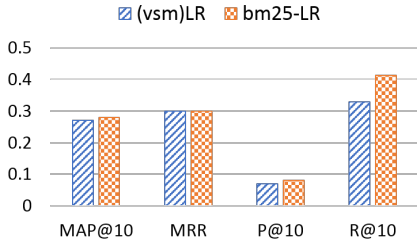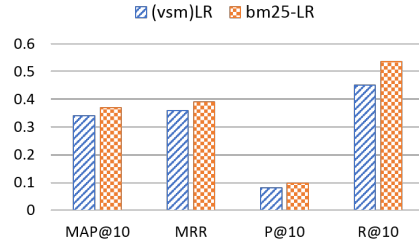
Fig. 2: Comparison with LR (**Birt**)

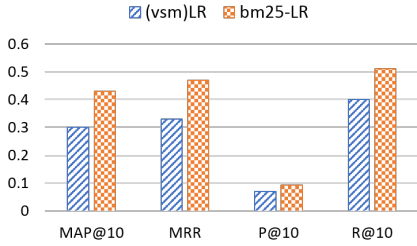Fig. 3: Comparison with LR (**Eclipse-UI**)
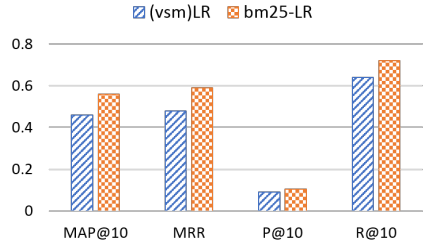

Fig. 4: Comparison with LR (**JDT**)

Fig. 5: Comparison with LR (**SWT**)

We observe that for all the projects our modification indeed helps to improve the overall performance in all the evaluation metrics. In JDT, SWT, and Eclipse-UI projects modified bm25-LR achieves a percentage performance gain of 43%, 21%, and 8% at metric MAP@10 (Figure 4, 5, and 3) respectively. On the other hand, we find that both the tools perform comparably for the Birt projects (Figure 2). For this project, we explore the feature weights and observe that the overall tool's performance heavily depends on the non-source code features (e.g., Class name, bug fix history, etc). Thus the superior performance of BM25 over VSM for source code similarity (as in RQ4) is less visible in the combined score. Similar results also observed previously by Ye et al. [8] on the same dataset. A similar conclusion can be drawn for other evaluation metrics.

These results indicate that with an informed choice of similarity measure a better result can be achieved for the bug localization task.

> **Result 5:** *A bug localization tool built with an informed choice of similarity measure can significantly improve the performance of original tool.*

5 Discussion

Our empirical study shows that the performance of different IR models varies with document types. Thus an informed model choice based on document type might help to provide a systematic way to take advantage of all the existing standard similarity measure for a better tool performance. For instance, it has

been observed that code search engines perform poorly to satisfy user's code related information need [44,45]. The inherent problem with such system is that to answer any code query, the search engine has to consult with a diverse set of documents (e.g., API documentation, StackOverflow post, GitHub issue, etc.). Thus, relying on a standard IR model might result in a very poor overall performance. In contrast, a systematic way of combining different optimal similarity measures might lead to a better search performance. Exploring this in details can be an interesting future work.

6 Related Work

Panichella et al. [46] propose a Genetic Algorithm based approach to automatically configure and assemble IR models. Oliveto et al. [47] empirically show the equivalence of different similarity measure for traceability recovery task. Gethers et al. [48] propose an integrated approach to combine orthogonal IR techniques: VSM, probabilistic Jensen and Shannon (JS) model [49], and Relational Topic Modeling (RTM) [50,48] for traceability recovery task. Evaluating on one repository (EasyClinic) containing 37 target/candidate documents, they analyze the impact of artifact types (i.e. use cases, UML diagrams, and test cases) on these IR models (i.e. JS, JS+RTM, VSM and RTM+VSM). They find that the combination with RTM is highly valuable when tracing with UML diagrams artifact.

Other parameters may also influence the performance of SE tasks. For example, incorporating user interaction also found to be effective in relevance feedback [51]. Type of query document also found to have an influence on similarity model choice [52], which is also confirmed by our finding. Researchers also propose heuristics based [38] and search-based [53,39] optimization techniques to calibrate IR methods for improved performance. Automatically learning weights while combining different IR models are also proposed in literature [54,8]. We complement these works by focusing on similarity model choice for different SE artifacts and demonstrate that an informed choice based on the document features can lead to a better performance.

7 Threats to Validity

From our experimental setup, some threats to the internal validity can arise. Apart from similarity measure, there are some other steps: preprocessing, stopword removal, stemming, etc. that can impact the performance [46]. We minimize the impact of this threat by applying similar techniques to all the considering models in each step. We also tune each model to its best performing configuration for the corresponding tasks to reduce any parameter configuration bias [38,39].

As GitHub hosts many open-source projects, our category dataset might not be representative enough. To increase the diversity in our project dataset, we use DMOZ Ontology [25], which is believed to represent the whole Web.

However, GitHub recently allows users to tag their projects. Though tags are not available for all projects, this tag information could be a possible alternative for DMOZ category. To curate the dataset for project recommendation, we manually annotated GitHub projects, which may impose unwanted bias. To mitigate this, we asked two annotators to annotate separately. We observed an agreement rate of 95% and resolved disagreement by discussion.

8 Conclusion

In conclusion, we performed an empirical study to understand the interaction between IR-based similarity measures and document types, and observed that model choice has a significant impact on performance for the different types of artifacts as well as mixtures of types. With this insight, we further investigate how such an informed choice of similarity measure impacts the performance of IR-based SE tasks. In particular, we analyzed two existing tools that leverage diverse SE artifacts. We found that such informed choice of similarity measure indeed leads to improved performance of the SE tools.

References

1. S. Haiduc, V. Arnaoudova, A. Marcus, G. Antoniol, in *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE'16)* (ACM, Austin, TX, USA, 2016), pp. 898–899
2. V. Arnaoudova, S. Haiduc, A. Marcus, G. Antoniol, in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2 (IEEE, 2015), vol. 2, pp. 949–950
3. G. Salton, A. Wong, C.S. Yang, A vector space model for automatic indexing, Communications of the ACM **18**(11), 613 (1975)
4. T.K. Landauer, S.T. Dumais, A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge., Psychological review **104**(2), 211 (1997)
5. M.J. Kusner, Y. Sun, N.I. Kolkin, K.Q. Weinberger, in *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)* (2015), pp. 957–966
6. V.J. Hellendoorn, P. Devanbu, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (ACM, 2017), pp. 763–773
7. S.E. Robertson, S. Walker, in *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval* (Springer-Verlag New York, Inc., 1994), pp. 232–241
8. X. Ye, R. Bunescu, C. Liu, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ACM, 2014), pp. 689–699
9. X. Ye, R. Bunescu, C. Liu, Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, IEEE Transactions on Software Engineering **42**(4), 379 (2016)
10. M. Wen, R. Wu, S.C. Cheung, in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on* (IEEE, 2016), pp. 262–273
11. J. Zhou, H. Zhang, D. Lo, in *Proceedings of the 34th International Conference on Software Engineering* (IEEE Press, 2012), pp. 14–24
12. H. Niu, I. Keivanloo, Y. Zou, Learning to rank code examples for code search engines, Empirical Software Engineering **22**(1), 259 (2017)
13. C. McMillan, M. Grechanik, D. Poshyvanyk, in *2012 34th International Conference on Software Engineering (ICSE)* (IEEE, 2012), pp. 364–374

14. C. McMillan, D. Poshyvanyk, M. Grechanik, in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering* (ACM, 2010), pp. 21–25

15. A. Marcus, A. Sergeyev, V. Rajlich, J.I. Maletic, in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on* (IEEE, 2004), pp. 214–223

16. X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, in *Proceedings of the 38th International Conference on Software Engineering* (ACM, 2016), pp. 404–415

17. T. Van Nguyen, A.T. Nguyen, T.N. Nguyen, in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on* (IEEE, 2016), pp. 749–751

18. B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, S. Li, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ACM, 2016), pp. 51–62

19. Z.S. Harris, Distributional structure, Word **10**(2-3), 146 (1954)

20. Adamhub. "`https://github.com/adamhub/screenbird`"

21. CrazyOrr. "`https://github.com/CrazyOrr/FFmpegRecorder`"

22. B.R.i.. of eclipse.platform.ui. "`https://bugs.eclipse.org/bugs/show_bug.cgi?id=369884`"

23. E.P. UI. "`http://projects.eclipse.org/projects/eclipse.platform.ui`"

24. C.V.E. of eclipse.platform.ui. "`https://github.com/eclipse/eclipse.platform.ui/blob/master/bundles/org.eclipse.e4.ui.workbench.swt/src/org/eclipse/e4/ui/internal/workbench/swt/E4Application.java`"

25. D.O. February 2017 Dump. " static mirror: `http://dmoztools.net/`. website: `http://www.dmoz.org/docs/en/rdf.html` "

26. GitHub. GitHub Search API. `https://developer.github.com/v3/search/`

27. Eid-Applet. "`https://github.com/e-Contract/eid-applet`"

28. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, D. Damian, in *Proceedings of the 11th working conference on mining software repositories* (ACM, 2014), pp. 92–101

29. Birt. "`https://www.eclipse.org/birt/`"

30. E. JDT. "`https://www.eclipse.org/jdt/`"

31. SWT. "`http://www.eclipse.org/swt/`"

32. A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, in *Proceedings of the 25th International Conference on Program Comprehension* (IEEE Press, 2017), pp. 218–229

33. M. Allamanis, E.T. Barr, C. Bird, C. Sutton, in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE'14)* (2014)

34. E.S. List. "`http://www.lextek.com/manuals/onix/stopwords1.html`"

35. J.L. Keywords. "`https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`"

36. M.F. Porter, An algorithm for suffix stripping, Program **14**(3), 130 (1980)

37. C.D. Manning, P. Raghavan, H. Schütze, *Introduction to Information Retrieval* (Cambridge University Press, New York, NY, USA, 2008)

38. L.R. Biggers, C. Bocovich, R. Capshaw, B.P. Eddy, L.H. Etzkorn, N.A. Kraft, Configuring latent dirichlet allocation based feature location, Empirical Software Engineering **19**(3), 465 (2014)

39. A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, in *Proceedings of the 2013 International Conference on Software Engineering* (IEEE Press, 2013), pp. 522–531

40. T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, in *Advances in neural information processing systems* (2013), pp. 3111–3119

41. Wikipedia. Wikipedia Dump. `https://dumps.wikimedia.org/enwiki/20160305/`

42. G. Gousios, in *Proceedings of the 10th Working Conference on Mining Software Repositories* (IEEE Press, Piscataway, NJ, USA, 2013), MSR '13, pp. 233–236

43. R. Řehůřek, P. Sojka, in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (ELRA, Valletta, Malta, 2010), pp. 45–50. `http://is.muni.cz/publication/884893/en`

44. M. Hucka, M.J. Graham, Software search is not a science, even among scientists, arXiv preprint arXiv:1605.02265 (2016)

45. S.E. Sim, M. Umarji, S. Ratanotayanon, C.V. Lopes, How well do search engines support code retrieval on the web?, ACM Transactions on Software Engineering and Methodology (TOSEM) **21**(1), 4 (2011)

46. A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1 (IEEE, 2016), vol. 1, pp. 314–325
47. R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on* (IEEE, 2010), pp. 68–71
48. M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia, in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (IEEE, 2011), pp. 133–142
49. A. Abadi, M. Nisenson, Y. Simionovici, in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* (IEEE, 2008), pp. 103–112
50. J. Chang, D.M. Blei, Hierarchical relational models for document networks, The Annals of Applied Statistics pp. 124–150 (2010)
51. G. Gay, S. Haiduc, A. Marcus, T. Menzies, in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* (IEEE, 2009), pp. 351–360
52. L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, A. Marcus, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ACM, 2015), pp. 567–578
53. S.W. Thomas, M. Nagappan, D. Blostein, A.E. Hassan, The impact of classifier configuration and classifier combination on bug localization, IEEE Transactions on Software Engineering **39**(10), 1427 (2013)
54. D. Binkley, D. Lawrie, in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on* (IEEE, 2014), pp. 441–445