Toward Optimal Selection of Information Retrieval Models for Software Engineering Tasks

Md Masudur Rahman
Department of Computer Science
Purdue University
West Lafayette IN, US
rahman64@purdue.edu

Saikat Chakraborty, Gail Kaiser, Baishakhi Ray
Department of Computer Science
Columbia University
New York NY, US
{saikatc, kaiser, rayb}@cs.columbia.edu

***Abstract*—Information Retrieval (IR) plays a pivotal role in diverse Software Engineering (SE) tasks, e.g., bug localization and triaging, bug report routing, code retrieval, requirements analysis, etc. SE tasks operate on diverse types of documents including code, text, stack-traces, and structured, semi-structured and unstructured meta-data that often contain specialized vocabularies. As the performance of any IR-based tool critically depends on the underlying document types, and given the diversity of SE corpora, it is essential to understand which models work best for which types of SE documents and tasks.**

We empirically investigate the interaction between IR models and document types for two representative SE tasks (bug localization and relevant project search), carefully chosen as they require a diverse set of SE artifacts (mixtures of code and text), and confirm that the models' performance varies significantly with mix of document types. Leveraging this insight, we propose a generalized framework, SRCH, to automatically select the most favorable IR model(s) for a given SE task. We evaluate SRCH w.r.t. these two tasks and confirm its effectiveness. Our preliminary user study shows that SRCH's intelligent adaption of the IR model(s) to the task at hand not only improves precision and recall for SE tasks but may also improve users' satisfaction.

***Index Terms*—Information retrieval; IR metrics; project recommendation; bug localization**

I. Introduction

Information retrieval (IR) plays a pivotal role in many Software Engineering (SE) tasks. Haiduc et al. [1] identified more than 20 different SE tasks, e.g., feature location, traceability link recovery, bug localization and triaging, that benefit from IR. IR techniques generally depend on three key components: (i) a *query* that expresses the user's information need; (ii) a corpus of *candidate documents* from which the relevant information is extracted; and (iii) an IR *model* that considers a query and a corpus of candidate documents, and computes a similarity score between the query and each candidate document. Typically, the candidate documents are then ranked by decreasing values of the similarity scores. Similarity scores based on similar bag-of-words (e.g., VSM [2], BM25 [3]) and context-based matching (e.g., LSI [4], WMD [5]) are some well-known IR models that measure document similarities.

When applying IR to SE tasks, well-established models such as the above are typically used as they are already stable, fine-tuned and well-explored, and perhaps SE researchers are primarily concerned with solving their SE problem rather than exploring alternatives among IR models. Although further tuning these models for SE tasks has been investigated [6], IR models have been developed mainly for natural language (NL) text corpora. But SE corpora, which often contain diverse document types including source code, test cases, bug reports, API documentation, project overviews, etc., are linguistically quite different from conventional natural language even when in text form [7]. For example, researchers have shown that Google web-search, whose IR models are heavily optimized for natural text, does not perform as well for code search [8].

Thus, we start this paper with a very simple question: among the available IR models, are there any models (or their combinations) that work better for a given SE task than others? In particular, we investigate whether choice among IR models has any significant impact on the SE task at hand—which model is suitable for source code elements (e.g., method names), which model is right for specific kinds of documents (e.g., bug reports), which model is more appropriate for tasks measuring the similarities across these two types of artifacts (e.g., for bug localization)? We devise a lightweight framework to automate the model selection, combination, and parameter tuning process, and show that tools built with such informed choices can outperform baseline tools by a significant margin.

We investigate two representative tasks—that often arise during software maintenance—where IR techniques have often been used in the past: (i) *bug localization*: given a bug report as query, retrieve the most likely to be relevant source file(s), so the developer can fix the bug [9]–[15], and (ii) *project recommendation*: given a GitHub project as query, find functionally similar GitHub projects, for the developer to explore "issues", test cases, reviews, and other artifacts that may grant insights that could lead to improving the developer's own project [16]–[22]. These tasks are carefully chosen to require using IR methods across multiple different kinds of SE document types. While bug localization relies on similarity computation between *heterogeneous* document types (code vs. bug report), for project recommendation similarity needs to be measured among multiple different but *homogeneous* types of documents (code vs. code, GitHub description vs. description, GitHub readme vs. readme, etc.).

We studied 1100 bug reports for bug localization and 1832 GitHub projects for project recommendation considering a range of different IR models. We found that BM25 performs best for the code vs. bug report of bug localization, but it is not as effective for project recommendation's homogeneous document comparison. For the various natural language comparisons of project recommendation, the LSI and WMD context-aware models worked better, while the keyword-based bag-of-words VSM model performs best for code vs. code.

Given the availability of many off-the-shelf IR models, it is challenging to choose the right one for these and other SE problems, particularly when any given model is likely to require tuning to achieve its optimal performance for the task and corpus at hand [23]–[25]. Moreover, for those SE tasks that involve different mixes of comparisons across documents, a single IR model may not be the best choice for all similarity comparisons. We constructed our generic framework, SRCH (**S**oftware Sea**rch**), to automatically select the optimal IR model, or set of models, for a given SE task.

We evaluate SRCH w.r.t. the two tasks introduced above: For project recommendation, SRCH recommends similar projects with a mean average precision of 76% for finding top-10 related projects (MAP@10). SRCH boosts the accuracy of project recommendation up to 24% w.r.t. baseline tools that use off-the-shelf VSM or LSI. SRCH also significantly outperforms two state-of-the-art tools: CLAN [16] and RepoPal [22] by 186% and 107%, respectively, at MAP@10. The BM25-based model selected by SRCH for bug localization outperforms the VSM-based baseline, achieving up to 43% performance gain, even though VSM has been previously been used in many previous bug localization tools, e.g., [12], [26]–[28].

We also conducted a preliminary user study with 12 users to evaluate SRCH w.r.t. project recommendation: these users found useful recommendations within top-5 outcomes (MAP@5) for 88% of the example queries.

This paper makes the following contributions:

1) We provide empirical evidence that the success of IR-based SE tasks depends significantly on the choice of IR model(s), which in turn depends on the underlying document type(s), and that careful selection, combination and tuning of IR models can improve the accuracy of IR-based SE tasks.
2) We propose and evaluate SRCH, our generic framework to automate IR model selection and tuning for SE tasks. We also show that SRCH can be used in legacy environments to improve accuracy.
3) We curate a valuable dataset of 1832 GitHub projects by retrieving their descriptions, `readme` contents, class and method names, imported package usage, and APIs for project recommendation, where we manually associated each project with a fine-grained category that describes its functionalities. Our dataset is publicly available at https://github.com/masud99r/IR-in-SE

II. Background

A. IR Models

1. Vector Space Model (VSM) [2] models represent documents (D) and queries (q) 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., $q = (qw_1, ..., qw_N)$ and $D = (Dw_1, ..., Dw_N)$ where the qw_i and Dw_i are the weights of the term i in a bag-of-words (BOW) 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 $sim(q, D) = \cos(q, D) = \frac{q^T D}{||q|| \ ||D||}$.

Implications: The VSM model is effective and simple to implement. However, as a BOW-based approach, it ignores token order. In the cosine similarity formula, the magnitudes of the document vectors ($||q||$ and $||D||$) are in the denominator and give smaller cosine values for larger dimensional vectors. Thus, longer documents may be penalized because they have more components that are indeed relevant.

2. BM25 [3] (BM stands for B̲est M̲atched, 25 refers to a standard encoding) is a different BOW model that looks for how many of the query terms are present in a document. It ranks the document with the highest number of query terms, normalized by document length, at the top.

Implications: A distinguishing feature of BM25 is that it treats a matching term's importance in the document and in the query differently, and also gives special attention to that term's frequency in the query. This improves performance when the query and document are of different types. Then document length normalization enables more accurate rank prediction when candidate documents are of various lengths. Despite these advantages, BM25 is (like VSM) a keyword-matching model that ignores word order. Thus BM25 might fit well where document length varies and token order doesn't matter.

3. Latent Semantic Indexing (LSI) [4] assumes 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 the raw co-occurrence count of a term. The IDF can be estimated from the document corpus. After inferring the lower dimensional vector of both query and candidate documents, cosine similarity computes the similarity between two document vectors as equation $sim(q, D) = \cos(q, D) = \frac{q^T D}{||q||||D||}$.

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

4. Word Embedding. This approach also assumes that similar words should have similar context [29]. In *Word Embedding*, a natural language processing (NLP) technique, each word w is represented by a d-dimensional vector of real numbers. This vector is learned from the *context* formed by the words preceding and following w in a sentence. Similar words should have similar context 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)* has proved to be the winner [5]. For each query term, WMD searches for the semantically closest term in each document, where the distance between two terms is calculated as a Euclidean distance in the word embedding space. The summation of the minimum distances for all query terms represents the distance from a query to a candidate document.

Implications: As word embedding captures the words' contextual information, WMD bridges 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 a keyword-based BOW model could not find any similarity between these two documents. In contrast, WMD can efficiently judge that they are highly similar since they have very similar word embeddings. In SE artifacts, synonymous terminology is common;e.g.,upgrade and update are often used interchangeably. WMD may be useful to detect similarity among documents with no identical words.

B. Studying SE Tasks

We analyze the effect of different IR models on different types of software documents w.r.t. two tasks that often arise during software maintenance:

(i) **Bug Localization.** Given a bug report as the query, this task ranks all the source files in the project repository based on their relevance to the query [9]–[15]. The files that top the ranking are more likely to contain the root cause of the bug. For example, for bug report id 369884 [30] in the Eclipse-Platform-UI [31] project, file *E4Application.java* [32] was fixed (see Table IV). A perfect bug localization tool would rank this file at top if queried with the above bug report.

(ii) **Project Recommendation.** During projects' evolution, developers often look for similar applications from which to port similar features [33], [34], explore relevant test cases and library usage [35], and look for other artifacts that may grant insights to improve the developer's own project [16]. 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 [16]–[22]. For example, screenbird [36] and FFmpegRecorder [37] are both Video Recorder software. For a query with the first project, the tool should return a list of Video Recorder projects that includes the second project (see Table VI).

III. Methodology

A. Study Subjects

We analyze a wide variety of projects for studying the two SE tasks introduced in Section II-B. For bug localization, Table Ia, we collect a benchmark bug report dataset [26], [38] that contains 1100 bug reports from four projects. For project recommendation, Table Ib, we use a total 1832 GitHub projects across 112 functional categories.

B. Data Collection

Collecting Bug Report Data. The bug report dataset, which has been used previously for bug localization in [26]–[28], [38], studies four projects: Birt [39], Eclipse Platform UI (Eclipse-UI) [31], Eclipse JDT [40], and SWT [41]. Each bug report contains a summary, description, report time, and status of its fix along with the bugfix commit. We downloaded the before-fix version of each project, and treated its files that were deleted or modified in the bugfix commit as the true buggy files. Any files added to the before-fix version cannot be predicted, so are not part of the evaluation.

TABLE I: **Study Subjects**

(a) **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

(b) **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

Collecting GitHub Projects. For studying project recommendation, we collect GitHub open source projects using the following two approaches: First, in (i) *Method-A*, we search GitHub with keywords representing project functionalities (e.g., media player, text editor, etc.), and download the relevant projects. However, as GitHub search primarily looks at project descriptions, a project without a proper description will not be retrieved in this step. Hence, in (ii) *Method-B*, we instead found GitHub projects with Google Play links and utilized their Google Play descriptions.

Method-A. Given a project functionality (e.g., Video Recorder), we use GitHub search API [42] to search for relevant projects using the functionality term as the search keyword. We select different types of functionalities using the DMOZ Ontology [43], 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 homonyms to reduce confusion of the search task. This approach gives us 90 different types of project functionalities.

We use these functionalities to retrieve different types of Java projects from GitHub. We exclude the forked projects as they include near-identical projects and overfit our project similarity data. For each query, we select the top $1,000$ projects with 3-star rating and above from the search results. We end up with 2180 unique projects under 90 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, project *Eid-Applet* [44]'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. This reduces to 1590 projects under 78 different functionalities.

Method-B. Here we collect 242 GitHub projects that have Google Play links in their descriptions or README contents but lack keywords about project functionalities. Again we exclude forked projects and projects with less than 3 stars to try to avoid toy projects [45]. Then we manually annotate these projects using the details available in Google Play, particularly the app description, similar app suggestion, category, etc.. We found some new project functionalities not seen in Method-A.

Finally, our dataset has 1832 (1590 + 242) projects with 112 different functionalities, shown in Table Ib, where Media Player, Search Engine, Database Systems, etc. are the top functionalities with the most member projects.

For both methods, two authors of this paper (separately) annotated by project category; they agreed in 95% of cases, and resolved disagreements by discussion. As the annotator needed to consult various documents (i.e. description, readme, Google store, etc.) to determine 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 approx. 90 working hours.

C. Feature Extraction

Features of the bug localization task. We extract six different features similar to Ye et al. [26]:

(i) Source Code: To begin with, we consider all the source files as a single document. Next, to compare with the baseline [26], we also implement another setting where only the method bodies of a source file are considered eliminating import statements, class names, etc.. We extract the method bodies using an Eclipse JDT [40].

(ii) API Description: To bridge the lexical gap between text and code, we leverage the API documentation. We build the document for a source file by concatenating the description of all the API classes used in the source files.

(iii) Collaborative Filtering Score: For a given bug report, the bug-introducing files may be the same files that were fixed before to fix similar prior bugs [46]. This feature prioritizes the previously fixed files. For each file, we build a document concatenating all the previous bug reports for which the file was responsible. The similarity is calculated between the query bug report and this constructed document.

(iv) Class Name: The existence of Class name tokens in a bug report is a strong indication that the corresponding Class file might be responsible for the bug. It is hypothesized that the longer the Class name, the stronger the signal of bugs [26]. This feature concatenates all the class names per file.

(v) Bug-Fixing Time: If a source file was fixed recently, it is more likely to contain bugs than a file that was fixed a long time ago [26]. This score is calculated by the inverse of the time difference in months between the query's bug reporting time and the most recent (previous) bug fix time of the corresponding source file.

(vi) Bug-Fixing Frequency: It is also assumed [26] that if a file has been responsible for fixing many previous bugs, it is more likely to contain bugs in future. Thus, this score is calculated as the number of bugs previously fixed in a file.

Features of project recommendation task. We choose five types of SE artifacts:

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

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

(iii) Method & Class names: Developers often use meaningful identifier names when implementing their project [47]. 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 retrieved method and class names that are declared within a project.

(iv) Import Package name: Similar projects often use similar API packages [16]. This motivates us to use imported API package names and class names as features. We use the Eclipse JDT [40] 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 assumed to be API names.

D. 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 characters (i.e. non-English) and punctuation. As a convention, Java uses camel case format for class, method, and variable names. For such compound tokens (e.g., *TerminalFactory*), in both text and code artifacts, we further extract smaller token units (i.e. *Terminal* and *Factory*). We also keep the original compound token to keep actual keyword information. We then normalize the tokens: remove numeric characters and convert to lower case letters. To avoid bias from the frequently occurring but less informative tokens we remove two types of stopword: standard English stopwords (adopted from [48]) and Java language related stopwords, i.e. keywords [49]: *void*, *public*, *while*, etc. To reduce the unwanted lexical gap between tokens, we apply the Porter Stemmer[50] to convert words to its base form (e.g., convert *computes* and *computed* into *comput*).

E. 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 match its corresponding ground truth. We use several standard evaluation metrics [51] 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 queries [51]. First, for each query, an average precision is computed for each rank. Given a query(q) and its ranking documents, the 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 the ranking position of the relevant document i in the retrieved ranking and $i/rank_i = 0$ if the 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. The 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.
5. Normalized Discounted Cumulative Gain (NDCG) is another popular metric for evaluating search-related tasks, emphasizing retrieving highly relevant documents [51]. Instead of binary judgment, relevant or irrelevant, higher value on a scale of $[0, r]$, where $r > 2$, indicates greater relevance.

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

F. Model Configurations

Performance of IR models varies significantly with different parameter settings [23], [25]. For a fair comparison, we tune each model to its best performing configuration for each task, shown in Table II. Since tuning is not the main focus of the paper, we simply did an exhaustive search varying the parameter values at regular intervals and chose the best.

TABLE II: **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 word2vec [52] word embedding model which is used by WMD. We use a diverse collection of $3.7M$ Wikipedia articles [53] for the training data. These articles usually contain multiple paragraph descriptions of Wikipedia concepts. Since a project description is usually shorter in length and might not be represented by the Wikipedia articles, we also include $7.5M$ GitHub project description collected using GHTorrent [54]. We also tried other word2vec models for import package and API, using corresponding documents for training, but found the resulting embedding less effective w.r.t. the project recommendation task. We use Gensim's [55] Python implementation of word2vec to train on our data.

For WMD in the bug localization task, we used a pre-trained word2vec model which is trained on source code and API documentation and found to be effective on the same dataset for bug localization [38].

IV. Empirical Study

Our central question is, for a given SE task, whether the choice of IR models matters significantly across different types of SE artifacts. In general, IR models are applied either on *heterogeneous* or *homogeneous* artifacts, depending on whether queries and documents are of different or similar types, respectively. The bug localization task is a classic example of the former where query and candidate document types are different (bug report vs. source code). In contrast, in project recommendation, IR models are applied to homogeneous artifacts: code vs. code, description vs. description, readme vs. readme, etc. We investigate the impact of different IR models: For each model, we choose its best performing configuration as shown in Table II.

RQ1. How well do different IR models perform across heterogeneous SE artifacts for the bug localization task?

TABLE III: **Impact of different IR models on the bug localization task using heterogeneous artifacts**

	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

Best performing values are highlighted in **Red** (**bold**) for each project

We check the similarities between bug reports and source files by studying 1100 bug reports from four projects. The query is the bug report and the documents are different source code files and meta-data. A successful IR model will rank the actual buggy file(s) at the top. Table III summarizes the results.

TABLE IV: **Sample results for bug localization**

Bug Reports and Fixed File	**Rank**
Bug 369884 [30] platform:/plugin/ not used for applicationXMI ... used for CSS resources or Icons. ... applicationXMI parameter. Also the e4 wizard should be adjusted to create the right URI. **Fixed File : E4Application.java** [32]	**BM25=1** VSM=31 LSI=110 WMD=5983

BM25 is the best performing model over all the models (tuned following Section III-F) across all the projects it outperforms other models significantly and achieves a percentage gain of (MAP@10, MRR): Birt (54%, 38%), Eclipse-UI (190%, 150%), JDT (366%, 287%), and SWT (320%, 266%),

TABLE V: **Impact of different IR models on project recommendation using homogeneous artifacts**

	Description				Readme				Method Class				Import Package				API			
	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD	VSM	LSI	BM25	WMD	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	**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	0.56	**0.61**	0.55	0.57	0.44	**0.45**	0.30	0.34	**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	0.40	**0.49**	0.41	0.33	0.25	**0.29**	0.14	0.14	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	0.13	**0.16**	0.13	0.10	0.07	**0.08**	0.03	0.03	0.06	**0.07**	0.02	0.03	**0.05**	0.03	0.01	0.02	**0.05**	0.04	0.04	0.04

Best performing values are highlighted in **Red** (**bold**) for each project

compared to the second best VSM model. For all individual queries these differences are statistically significant (p-value $\leq$ 0.05) with medium to large Cohen's d effect size [56]. Table IV shows an example, where BM25 ranks the intended file at top-1 position, whereas VSM ranks it at 31^{st} place.

Discussion. Our documents are source code files and the query is a bug report. Since source code files can vary significantly in length, cosine similarity methods like VSM and LSI inadvertently hurt longer documents as the document length $||D||$ is in the denominator of the cosine function (see Section II). BM25 mitigates this with document length normalization.

Moreover, VSM and LSI assume both query and document are of same type and represent them in the same conceptual space. For heterogeneous artifacts, representing them in the same space is not necessarily effective. Further, context-aware models like LSI and WMD are less effective—the same term can exist in different surroundings across two different types of documents. In the example of Table IV, the term "e4", which is one of the most important terms to associate the report with the file, appears in very different contexts: in the query "e4" appears in the bug report text, while in the source file document it is the class/file name.

Result 1: *BM25 achieves best performance for measuring similarities across heterogeneous documents for the bug localization task.*

RQ2. How well do different IR models perform for homogeneous SE artifacts for project recommendation?

Here, we study two types of SE artifacts: project documentation (description and readme text) and source code (method and class names, import package names, and API names).

For a given project set N, we take one project as the query and consider the remaining $N-1$ projects as candidate documents. If the categories (i.e. functionalities) 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 query set to get overall performance. We apply all four IR models to each feature and report the results for 200 query projects in Table V. We discuss the results mostly w.r.t. $MAP@10$. However, a similar conclusion can be drawn for all other evaluation metrics.

For the textual artifacts, in general, LSI achieves highest performance for most of the evaluation settings. For *project description text*, the LSI model performs best for all the evaluation metrics and gains 11.76% compared to the second best models (WMD, VSM, and BM25 perform comparably) at $MAP@10$. Similarly, for *readme files*, the LSI model performs best in most of the evaluation metrics —4.85% improvement for $MAP@10$ w.r.t. VSM, the second best model. Note that all the models are first tuned following Section III-F to their best-performing configurations.

TABLE VI: **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** ...

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

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

Discussion. Among the 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. [47] that developers use meaningful identifier names while writing a software program.

Both the context-aware models LSI and WMD show a similar decreasing trend in performance going from project document artifacts to code artifacts (see Table V). This result indicates that context sensitivity, which is quite effective in natural language text, is not that helpful for source code artifacts. Further, among source code artifacts, LSI and WMD perform better for method and class names, suggesting context is more meaningful for these kinds of names than import packages and API names. Interestingly, all the projects retrieved by WMD in Table VI are wrong, and they have few keywords overlapping with the query project. However, a closer look will reveal the document and query terms are related. This indicates GitHub projects descriptions are not very complex. We may not need a word embedding-based similarity metric, where contextual similarity plays a pivotal role, and may undermine simple keyword-based matching.

Except for the description feature, BM25 performs significantly worse for the rest of the artifacts. BM25 treats query and document differently (see scoring equation in Section II), although for homogeneous artifacts the query and documents are linguistically identical. BM25 also assumes all the terms in the query are important and does not normalize w.r.t. query length. For example, in Table VI, all except the top project are not correct for BM25. As it ignores inter-relationships among query terms, BM25 emphasizes all parts of the query equally so is misguided by the query's variety of concepts. Thus, a verbose query may hurt BM25 performance.

Result 2: *For the homogeneous SE artifacts of the project recommendation task, the context-aware LSI model performs better for textual artifacts while the simple Bag-Of-Word based VSM outperforms others for code artifacts.*

The empirical study demonstrates that different IR models, or combinations of models, do better than others for different SE tasks involving different mixes of document types.

However, with many different IR models and their number of available tuning parameters, the search space of finding a suitable IR model (or combination of models) is quite large, so it is non-trivial to find the optimal one. To facilitate the tool builder, we devise a light-weight framework that automatically selects the right model(s) for a given SE task.

V. SRCH FRAMEWORK

For a given SE task, SRCH takes as inputs a query and a set of documents on which the query will operate, and a potential set of IR models that SRCH will explore. Each document is associated with some document features $(F_1, ..., F_n)$. For example, for bug localization task bug_report is a query and source code, API description, etc. are the document features. At a high-level, for each feature (F_i), SRCH selects an optimal performing model and then aggregates all the chosen models to generate the final model. The final model associates a similarity score to each potential query-result candidate. The final output is a ranked list of these candidates based on the final score (S) attributed to them.

Thus, for a given candidate output (τ), SRCH generates score linearly combining all the per-feature models, i.e.,

$$S=\sum_i \lambda_i \hat{M}_i(F_i,\tau), \text{ where } \sum_i \lambda_i=1 \qquad (1)$$

where, $\hat{M}_i$ is the output of *optimal performing* IR model for feature F_i, i.e.,

$$\hat{M}_i=\text{argmax}_{m_j \in M}\{m_j(F_i,\tau)\} \qquad (2)$$

SRCH involves two main steps: (i) *Parameter Tuning.* To achieve an optimal performance SRCH empirically tunes the underlying parameters using a set of training data. The training data contains queries, documents, and ground truth results (e.g., true buggy files for bug localization task). First, for each document feature, SRCH selects the optimal performing model using equation 2. Note that all the models are tuned with their best-performing configuration before choosing the best one. Next, SRCH linearly combines the output scores of per-feature optimal model with an weighted average, as shown in equation 1. SRCH empirically selects weights (λ_i)'s based on training data satisfying the constraint $\sum_i \lambda_i=1$.
(ii) *Ranking Search Results.* With the tuned final model, SRCH assigns scores to each potential query-result candidate (τ_1, ..., τ_m). For each of the document τ_j, SRCH generates score S_j with equation 1. Then SRCH sorts the target documents based on their corresponding scores.

A. Framework Evaluation

RQ3. Can bug localization be improved using the SRCH framework?

TABLE VII: **Optimal models for different features identified by SRCH framework for the bug localization task**

	Source Code	API Descr.	Collab. Filter	Class Name	Bug Fix Time	Bug Fix Freq
BugSrch	BM25	BM25	BM25	N/A	N/A	N/A
BugSrch-VSM (baseline)	VSM	VSM	VSM	N/A	N/A	N/A

SRCH systematically combines the optimal IR models for bug localization features and produces a final model, BUGSRCH. Among the six features used in this task (see Section III), three features—Class Name, Bug fix time, Bug fix frequency—are meta information that do not require any IR model. We follow a similar technique to Ye et al. [26] to compute the other feature values for both BUGSRCH and a baseline we call BUGSRCH-VSM, which uses only the VSM model (see Table VII). In RQ1 we observed that VSM is the second best performing model on the bug-report dataset. Thus, we only report and compare BUGSRCH w.r.t. BUGSRCH-VSM here. Additionally, VSM has been popularly used in many previous bug localization tools (e.g., [12], [26]–[28]).

Compare with State-of-the-art. We also compare BUGSRCH results with a state-of-the-art bug localization tool, LR, proposed by Ye et al. [26]. LR is methodologically similar to our BUGSRCH as both use the same feature set to build a combined model. LR also leverages a learning to rank model to compute optimal feature weights, to generate a combined model score. Thanks to Ye et al. [26] providing us the ranked results for the original LR model, we could compute the reported results of LR [26] for the same test set. For this evaluation, we use the latest 200 bug-reports for JDT and 100 each for three other projects from the benchmark dataset of Section III. The performance of the combined model heavily depends on the assigned weights (λs). To mitigate such impact, we calibrate the weights and report the best performance for both BUGSRCH and BUGSRCH-VSM.

TABLE VIII: **Percentage gain of BUGSRCH over baseline and state-of-the-art tool. A positive value indicates an improvement**

	Gain over BUGSRCH-VSM				Gain over LR			
	MAP@10	MRR	P@10	R@10	MAP@10	MRR	P@10	R@10
Birt	3.70	0.00	15.94	25.85	75.00	76.47	86.05	56.07
Eclipse-UI	8.82	8.33	16.67	18.01	12.12	11.43	55.56	14.44
JDT	43.33	42.42	39.10	28.32	2.38	4.44	-3.65	-9.31
SWT	21.74	22.92	13.98	11.96	27.27	28.26	12.77	8.80

Table VIII shows that the performance gain of BUGSRCH over BUGSRCH-VSM and LR [26] in different evaluation metrics for the different projects. A positive gain value indicates an improvement. We see BUGSRCH outperforms BUGSRCH-VSM for all the projects, achieving up to 42% and 43% gain at MRR and MAP values, respectively. Compared to the state-of-the-art tool LR, BUGSRCH achieves improved performance in most of the evaluation metrics for all the projects, achieving up to 75% MAP, 76% MRR gain, and 86% better precision.

As noted, the LR tool combines the weights of different features using a learning to rank method [26]. In contrast, BUGSRCH uses the weight-tuning approach. Thus, the only difference between BUGSRCH-VSM and LR is how they determine the combined weights, as both the tools use VSM as their underlying IR model. The results of Table VIII also indicate that a good weight-tuning might be better than a learning-to-rank approach. Nonetheless, BUGSRCH outperforms both tools, showing the effectiveness of our SRCH framework.

Result 3: *The informed combination of IR models built by* SRCH *significantly improves the bug localization performance and significantly outperforms the example baseline and state-of-the-art tools.*

RQ4. Can project recommendation be improved using the SRCH framework?

For project recommendation, SRCH generates a combined model, which we call PROJSRCH.

Baseline Selection. To evaluate the effectiveness of PROJSRCH (the combined model generated by SRCH), we build two baseline tools: (i) PROJSRCH-LSI: uses only the LSI model for all features, and (ii) PROJSRCH-VSM: uses only the VSM model for all features. We choose LSI and VSM as they are popular IR models and used by previous project recommendation tools [16], [19], [22]. We use the same weights as PROJSRCH for these baselines where they also achieve optimal performance. Table IX shows the model assignments in detail.

TABLE IX: **Optimal models for different features identified by SRCH framework for the project recommendation task**

	Feature				
	Description	Readme	Method-Class	Package	API
ProjSrch	LSI	LSI	VSM	VSM	VSM
ProjSrch-LSI	LSI	LSI	LSI	LSI	LSI
ProjSrch-VSM	VSM	VSM	VSM	VSM	VSM

State-of-the-art tools selection. We also compare PROJSRCH with two state-of-the-art project recommendation tools: CLAN [16] and RepoPal [22]. CLAN compares JDK APIs (packages and classes) used in the studied projects using the LSI algorithm to establish similarities. Since CLAN's source code is not available, we reimplemented CLAN adhering to the paper details. We further extended CLAN to incorporate all the APIs studied by PROJSRCH, for a fair comparison. We also tune the feature weights and report the best results at weight *import package* $= 0.9$ and *API* $= 0.1$.

We find that VSM is the best performing metric for the features used by CLAN (see RQ1). Thus, we build a modified version of CLAN, PROJSRCH-vsmCLAN, where we replace the similarity metric used by CLAN with VSM. We also tune the weights and report the best results at weights *import package* $= 0.6$ and *API* $= 0.4$.

The other state-of-the-art tool RepoPal [22] uses the GitHub project popularity metric *star-count* and the readme content to detect similar projects. They assume that projects starred by the same user within a short period are similar. Thus, they calculate *star-relevance* between two projects. They also calculate a *readme-relevance* score based on the readme contents of the two projects using VSM, and combine with the *star-relevance* score to get the final similarity score. We use the publicly available *star-relevance* implementation of RepoPal to get the *star-relevance* score. As the other part of their system is not available, we follow the paper's descriptions to reimplement the *readme-relevance* module and the combined model.

Note that the omission of any meaningful feature in any project might hurt the performance of the model that uses that feature. As different tools are using different combination of features we exclude the projects we annotated using method-B as this set might contain projects with a missing feature (e.g., description). We also exclude the projects used as the query in RQ1 to avoid any model selection bias for PROJSRCH. Finally, we select a set of 1590 projects and report the results for a query set of 1390 projects.

Results. Table X shows the percentage gain of PROJSRCH over baseline tools PROJSRCH-LSI and PROJSRCH-VSM. We see that PROJSRCH outperforms both the baseline tools in all evaluation metrics, achieving a performance gain ranging from 4.88 to 24.79. This shows that an informed model choice for feature artifacts boosts the performance of PROJSRCH.

TABLE X: **Percentage gain of PROJSRCH over baselines**

	MAP@10	MRR	P@10	R@10
PROJSRCH-LSI	6.56	4.88	13.34	17.56
PROJSRCH-VSM	12.58	9.38	21.51	24.79

We also compare the effectiveness of these models w.r.t. two state-of-the-art tools, CLAN [16] and RepoPal [22], on the same query set. In Figure 1, we see that ProjSrch outperforms the previous tools—it achieves 186%, and 162% gain over CLAN for MAP@10 and MRR measures, respectively, and w.r.t. RepoPal, ProjSrch achieves 107% and 97% gain for MAP@10 and MRR measures.

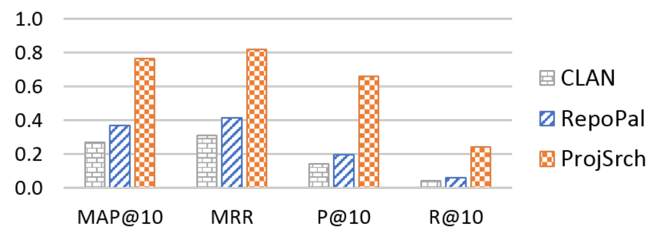

Fig. 1: Comparison with state-of-the-art tools

Srch systematically selects the best similarity metrics for each feature and generates a combined similarity score. These results show that the model selection helps ProjSrch to improve over RepoPal and CLAN. In addition, ProjSrch extracted useful features from GitHub projects, which might play a role to boost ProjSrch's performance. Unlike the other tools, RepoPal uses an additional *star-relevance* feature, which boosts its performance [22].

On the other hand, CLAN uses only import package and API. To compare them on the same ground, we further evaluate the impact of model selection and restrict ProjSrch to use the same feature set as CLAN. We name this version as ProjSrch-vsmCLAN. Srch chose VSM for these features as it is the best performing one. We observe that, even with the same feature set, ProjSrch-vsmCLAN performs better than CLAN: 26 to 35 percent performance gain is achieved under different evaluation metrics.

All these results strongly suggest that, with the right choice of IR model per feature, a project recommendation tool can outperform its predecessors.

Result 4: *A project recommendation tool built by* Srch *framework with an informed choice of IR models can significantly outperform both baselines and state-of-the art tools.*

B. Informal User Study

We asked human evaluators for their opinions about the performance of the ProjSrch generated by Srch. For each query project, we provided the top 10 ranked projects retrieved by ProjSrch, and for all projects, we provided the project's name, description, and associated GitHub URL. We asked the user to give a relevance score on a scale of 0 to 5 (where 0 means not relevant, and 5 means most relevant) for each of the retrieved results considering functional similarities between query projects and corresponding retrieved projects. We instructed the user to look into the project's readme file, source code, etc. on Github in addition to looking at the project name and description. We also asked the user to give a partial score if two projects are partially similar. We selected 25 queries from our project dataset and gathered relevance judgment scores from 12 users (CS Professionals(3), non-CS Professional(1), and CS Grad Students(8)).

ProjSrch achieved evaluation scores 0.88 and 0.84 for $NDCG$@5 and $NDCG$@10, resp. Intuitively, in 88% of the cases users encountered relevant (satisfactory) results when they examined the top 5 returned documents. Higher NDCG values indicate ProjSrch, hence the Srch framework, is deemed useful to the users. One user commented after using our tool: "I can usually find a relevant app/project in your top 3 results, which is impressive". Another user said: "The most interesting case I found is clickerfree under the calculator category. This app does not have a highly relevant terms like calculator, but it indeed has the calculation functionality, which may not be found by the existing search techniques."

VI. Study Implications

(i) **Generalization of Srch**. To demonstrate the effectiveness of Srch we use the four IR models and discussed them for the two SE tasks. However, any new models and tasks can be plugged in as needed. Assuming the availability of training data, any machine learning-based approach can be incorporated into the Model Combination module. Srch's individual metric score calculations can be done in parallel or distributed settings to reduce clock time.

(ii) **Implications for Code Search**. Beyond the Software Maintenance tasks presented here, Srch could also be leveraged in general purpose code search. It has been observed that current code search engines perform poorly [8], [57], [58]. The inherent challenge of such systems is that the search engine has to consult a diverse set of documents (e.g., API documentation, StackOverflow posts, GitHub issues, etc.), where relying on a single IR model might result in a worse overall performance than might be achieved by combining models. In contrast, Srch provides a systematic way to leverage all available similarity metrics.

(iii) **Implications of Empirical Findings**. RQ1 and RQ2 characterize the interactions between four IR models and diverse SE document types. Such characterizations can be extended to other mode models and document types as well. Our results can be leveraged by other SE tasks, such as bug triaging, that depend on the studied document types [59].

VII. Related Work

Comparison and Combination of IR models for SE tasks. Researchers have previously empirically evaluated different models, e.g., Gethers et al. [60] proposed an integrated approach to combine orthogonal IR techniques—VSM, the probabilistic Jensen and Shannon (JS) model [61], and Relational Topic Modeling (RTM) [60], [62]—for the traceability recovery task [63], [64]. Evaluating on one repository (EasyClinic) containing 37 target/candidate documents, they analyzed the impact of artifact types (i.e. use cases, UML diagrams, and test cases) on multiple combinations of IR models (JS, JS+RTM, VSM and RTM+VSM). They found that combination with RTM is highly valuable when tracing with the UML diagrams artifact. In contrast, we propose a generic framework and evaluate it on much larger data sets consisting of 1832 projects and

1100 bug reports, respectively, for two different SE problems, where we also evaluate each of the individual models on each type of artifact separately. We further characterize the properties of the artifacts that might influence the models' performance. Thus we confirm their findings, but at a much larger scale for different IR models, document types and tasks.

The type of the query document has previously been found to influence best choice of similarity metric [65], also confirmed by our findings. Other factors that we have not studied yet may also influence the performance of SE tasks. For example, incorporating user interaction has been found to be effective in relevance feedback [66].

Panichella et al. [6] propose a Genetic Algorithm-based approach to automatically configure and assemble IR models. Other researchers propose heuristics-based [23] and search-based [24], [25] optimization techniques to calibrate IR models for improved performance. Automatically learning weights while combining different IR models has also been proposed [26], [67]. We complement these works by focusing on similarity metrics choice for different SE document artifacts and demonstrate that an informed choice based on the document features can lead to better performance.

Project Recommendation. Some prior researchers used code clone analysis to establish project similarity [68]–[71], designed to identify plagiarized apps. In contrast, we are interested in conceptual or functional similarities between projects. Researchers also considered other collections of code-related features such as API package and class names [16], identifier names [17], method names, class names and code comments [18] to establish project similarities. Other available metadata has been used to find similar applications: collaborative tagging [19], GUI layout [20], API call intents, permissions, and sensors [21], project popularity [22], etc.. In contrast, we perform an in-depth analysis of the role of each sub-component and the different similarity metrics in determining project similarities. Thus, we complement this prior work. We have also empirically shown that an informed combination of IR models and SE artifacts can outperform a state-of-the art project recommendation tool.

Bug Localization. Many researchers have studied bug localization using IR techniques [9]–[15]. Zhang et al. [72] presents a survey. Proposed approaches to improve the bug localization task include combining bug reports' metadata [15], [26], modifying standard IR models [12], identifying similar bug fixes [12], [26], bug fix history [26], [73], source code metadata [73], combining topic models in a ranking metric [74], program spectrum [75], software changes [76]. In contract, we leverage the best performing similarity metrics on bug reports and source code artifacts to improve this task, demonstrating the use of SRCH rather than focusing on tool development.

Deep learning-based models DNNLOC [28], HyLoc [77] have been found effective compared to LR [26], but deep learning models come with huge cost (time and resources). In contrast, we demonstrate that even a simple informed choice of similarity metric can lead to significant improvement.

Researchers have also compared different IR models for the bug localization task and found that simple text models perform better than a topic model-based approach [10]. BM25-based models were also found to be effective for finding duplicate bug reports [78] and the bug localization task [79]. Our experimental results confirm their findings.

VIII. Threats to Validity

From our experimental setup, some threats arise to internal validity. Apart from similarity metric, there are some other steps: preprocessing, stopword removal, stemming, etc. that can impact performance [6]. We minimize the impact by applying similar techniques to all considered models in each step. We also tune each model to its best performing configuration to reduce parameter configuration bias [23], [25].

Due to the unavailability of previous tools, we re-implemented CLAN [16] and part of Repopal [22]. To mitigate this threat, we confirmed our implementation by cross-checking results with the reported results [22].

As GitHub hosts many open-source projects, our datasets might not be representative, a threat to external validity (generalizability). To increase the diversity in our project dataset, we use the DMOZ Ontology [43], 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 can be a possible alternative for DMOZ category. To curate the dataset for project recommendation, we manually annotated GitHub projects. To mitigate bias, two annotators worked separately, and then reached consensus.

A further threat to external validity is our evaluation on only two kinds of SE tasks. Future work will explore additional tasks amenable to IR models.

IX. Conclusion

We argue that the SE community should not blindly use any or even the best state-of-the-art IR model devised for conventional natural language text. Instead, we should choose carefully among competing IR models.

This work presents an in-depth empirical study to understand the interaction between IR models and SE artifacts. We found that an SE task's mix of similarity comparisons, between documents of the same or different types, has a significant impact on the performance of different IR models. Further, composing different models for different comparisons required by the same task may be better than tuning a single model. With this insight, we developed SRCH, a framework to automatically select and compose IR models for the mix of document type comparisons appropriate for the SE task at hand. We evaluate SRCH and confirm its effectiveness on two representative SE tasks selected to show different mixes of document type comparisons, where our approach outperforms baseline and state-of-the-art tools.

Acknowledgments

Ray is funded in part by NSF CCF-1619123, CNS-1842456, CNS-1618771, and CCF-1822965. Kaiser is funded in part by NSF CNS-1842456, CCF-1815494 and CNS-1563555.

References

[1] S. Haiduc, V. Arnaoudova, A. Marcus, and G. Antoniol, "The Use of Text Retrieval and Natural Language Processing in Software Engineering," in *38th International Conference on Software Engineering (ICSE)*. Austin, TX, USA: ACM, 2016, pp. 898–899.

[2] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[3] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[4] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." *Psychological review*, vol. 104, no. 2, p. 211, 1997.

[5] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, 2015, pp. 957–966.

[6] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms," in *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 314–325.

[7] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *11th Joint Meeting on the Foundations of Software Engineering (FSE)*. ACM, 2017, pp. 763–773.

[8] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K. T. Stolee, and B. Ray, "Evaluating how developers use general-purpose web-search for code retrieval," in *15th International Conference on Mining Software Repositories (MSR)*. ACM, 2018, pp. 465–475.

[9] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 155–164.

[10] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *8th Working Conference on Mining Software Repositories (MSR)*. ACM, 2011, pp. 43–52.

[11] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[12] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *34th International Conference on Software Engineering (ICSE)*. IEEE Press, 2012, pp. 14–24.

[13] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *22nd IEEE/ACM international conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 433–436.

[14] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *10th Working Conference on Mining Software Repositories (MSR)*. IEEE Press, 2013, pp. 309–318.

[15] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[16] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 364–374.

[17] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.

[18] A. Michail and D. Notkin, "Assessing software libraries by browsing similar classes, functions and relationships," in *21st International Conference on Software Engineering (ICSE)*. IEEE, 1999, pp. 463–472.

[19] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," in *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 600–603.

[20] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 1025–1035.

[21] M. Linares-Vásquez, A. Holtzhauer, and D. Poshyvanyk, "On automatically detecting similar android apps," in *24th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[22] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li, and J. Sun, "Detecting similar repositories on github," in *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2017, pp. 13–23.

[23] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[24] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[25] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 522–531.

[26] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 689–699.

[27] ——, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2016.

[28] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *25th International Conference on Program Comprehension (ICPC)*. IEEE Press, 2017, pp. 218–229.

[29] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[30] B. R. i. . of eclipse.platform.ui, ""https://bugs.eclipse.org/bugs/show_bug.cgi?id=369884"."

[31] E. P. UI, ""http://projects.eclipse.org/projects/eclipse.platform.ui"."

[32] 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"."

[33] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE 2012, pp. 53:1–53:11.

[34] B. Ray, C. Wiley, and M. Kim, "Repertoire: A cross-system porting analysis tool for forked software projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE 2012, pp. 8:1–8:4.

[35] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[36] Adamhub, ""https://github.com/adamhub/screenbird"."

[37] CrazyOrr, ""https://github.com/CrazyOrr/FFmpegRecorder"."

[38] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 404–415.

[39] Birt, ""https://www.eclipse.org/birt/"."

[40] E. JDT, ""https://www.eclipse.org/jdt/"."

[41] SWT, ""http://www.eclipse.org/swt/"."

[42] GitHub, "GitHub Search API," https://developer.github.com/v3/search/.

[43] D.-O. February 2017 Dump, "" static mirror: http://dmoztools.net/. website: http://www.dmoz.org/docs/en/rdf.html "," February, 2017 Dump.

[44] Eid-Applet, ""https://github.com/e-Contract/eid-applet"."

[45] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 92–101.

[46] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *35th International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 332–341.

[47] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[48] E. S. List, ""http://www.lextek.com/manuals/onix/stopwords1.html"."

[49] J. L. Keywords, ""https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html"."

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

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

[52] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[53] Wikipedia, "Wikipedia Dump," https://dumps.wikimedia.org/enwiki/20160305/.

[54] G. Gousios, "The ghtorrent dataset and tool suite," in *10th Working Conference on Mining Software Repositories (MSR)*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.

[55] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[56] M. E. Rice and G. T. Harris, "Comparing effect sizes in follow-up studies: Roc area, cohen's d, and r," *Law and human behavior*, vol. 29, no. 5, pp. 615–620, 2005.

[57] M. Hucka and M. J. Graham, "Software search is not a science, even among scientists," *arXiv preprint arXiv:1605.02265*, 2016.

[58] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 1, p. 4, 2011.

[59] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *26th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[60] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 133–142.

[61] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *16th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2008, pp. 103–112.

[62] J. Chang and D. M. Blei, "Hierarchical relational models for document networks," *The Annals of Applied Statistics*, pp. 124–150, 2010.

[63] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *18th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2010, pp. 68–71.

[64] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 378–388.

[65] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus, "Query-based configuration of text retrieval solutions for software engineering tasks," in *10th Joint Meeting on the Foundations of Software Engineering (FSE)*. ACM, 2015, pp. 567–578.

[66] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *25th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 351–360.

[67] D. Binkley and D. Lawrie, "Learning to rank improves ir in se," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 441–445.

[68] J. Crussell, C. Gibler, and H. Chen, "Scalable semantics-based detection of similar android applications," in *Proc. of Esorics*, vol. 13. Citeseer, 2013.

[69] ——, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[70] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 175–186.

[71] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.

[72] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, 2015.

[73] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 263–272.

[74] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[75] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *10th Joint Meeting on the Foundations of Software Engineering (FSE)*. ACM, 2015, pp. 579–590.

[76] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[77] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[78] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 253–262.

[79] Z. Shi, J. Keung, and Q. Song, "An empirical study of bm25 and bm25f based feature location techniques," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, ser. InnoSWDev 2014. New York, NY, USA: ACM, 2014, pp. 106–114.