From Word Embeddings To Document Similarities for Improved Information Retrieval in Software Engineering

Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu
School of Electrical Engineering and Computer Science, Ohio University
Athens, Ohio 45701, USA
xy348709,hs138609,xm108210,bunescu,liuc@ohio.edu

ABSTRACT

The application of information retrieval techniques to search tasks in software engineering is made difficult by the lexical gap between search queries, usually expressed in natural language (e.g. English), and retrieved documents, usually expressed in code (e.g. programming languages). This is often the case in bug and feature location, community question answering, or more generally the communication between technical personnel and non-technical stake holders in a software project. In this paper, we propose bridging the lexical gap by projecting natural language statements and code snippets as meaning vectors in a shared representation space. In the proposed architecture, word embeddings are first trained on API documents, tutorials, and reference documents, and then aggregated in order to estimate semantic similarities between documents. Empirical evaluations show that the learned vector space embeddings lead to improvements in a previously explored bug localization task and a newly defined task of linking API documents to computer programming questions.

CCS Concepts

•Information systems → Multilingual and crosslingual retrieval; Similarity measures; Learning to rank; •Software and its engineering → Software testing and debugging; •Computing methodologies → Lexical semantics;

Keywords

Word embeddings, skip-gram model, bug localization, bug reports, API documents

1. INTRODUCTION AND MOTIVATION

Recently, text retrieval approaches have been applied to support more than 20 Software Engineering (SE) tasks [11, 24]. In these approaches, the lexical gap between user queries and code [27] is usually identified as significant impediment. In feature location and bug localization [25], for example, the information need may be expressed as an issue report written in natural language, whereas the functions and files to be retrieved mainly contain code. In automatic code search [27], there is a mismatch between the high-level intent in user queries and the low-level implementation details. In community Question Answering (cQA), questions and answers are expressed in natural language, code, or a combination of both [6, 34, 40]. Consequently, standard information retrieval (IR) methods based on Vector Space Models (VSM) are plagued by the lexical gap between the natural language used in the user query and the programming language used in the source code [32]. Even when the query and the source files use a mix of statements in natural language and programming language (code snippets in queries and natural language comments in code), the performance of the IR system is sub-optimal due to the significant language mismatch.

An example of language mismatch in the bug localization task is illustrated in Figures 1 and 2. Figure 1 shows a bug report[1] from the Eclipse project, in which the author reports an abnormal behavior of the view icon when the view is minimized. Correspondingly, the bug report contains relevant keywords such as *view*, *icon*, and *minimize*. The bug is later fixed by changing the source file PartServiceImpl.java[2], which contains the code managing view behaviors. However, the code does not contain any of the keywords found in the bug report. Instead, the code contains keywords such as *stack* and *placeholder*. Overall, because the code and the bug report have no words in common, their cosine similarity in the standard $tf.idf$ vector space models used in IR would be 0. Nevertheless, it can be determined automatically that the two sets of keywords are semantically related through the use of the *distributional hypothesis* [12]. According to this linguistic hypothesis, words that appear in the same contexts tend to have similar semantic meanings. Figures 3, 4, and 5 show 3 types of documents where keywords from both the bug report and the source code appear in the same context. Figure 3 shows a fragment from the Eclipse user guide[3] that talks about minimizing views. In this fragment, the two types of keywords appear together in the same sentences. This is further illustrated in Figure 4, which shows a fragment from the plug-in developer guide[4], and in Figure 5 that contains a fragment from an API document. In fact,

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884862

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=384108
[2] https://git.eclipse.org/r/#/c/20615/
[3] http://help.eclipse.org/luna/nav/0
[4] http://help.eclipse.org/luna/nav/46

Bug ID: 384108
Summary: JUnit view icon no longer shows progress while executing tests
Description: Before I upgraded to Juno this morning I used to happily run tests with the JUnit view minimized, and enjoy seeing the progress of the tests on it. Now I don't see any change on the icon until if passes (where a green check appears) or fails (where a red X appears) ...

Figure 1: Eclipse bug report 384108, with keywords in red.

```
public class PartServiceImpl implements EPartService {
    ...
    private void recordStackActivation(MPart part) {...
        MPlaceholder placeholder = part.getCurSharedRef();
    ... }
    ...
    private void adjustPlaceholder(MPart part) {...
        MPlaceholder placeholder = part.getCurSharedRef();
    ... }
... }
```

Figure 2: Code from PartServiceImpl.java, with keywords in blue.

Section: Working with views and editors
Topic: Maximizing and minimizing in the eclipse presentation
Content: The minimize behavior for the Editor Area is somewhat different; minimizing the Editor Area results in a trim stack containing only a placeholder icon representing the entire editor area rather than icons for each open editor ...

Figure 3: Text from Eclipse Workbench User Guide.

Section: Making UI contributions
Topic: Adding the perspective
Content: The browser perspective defines two views (one visible, with a placeholder for the other) ...

Figure 4: Text from Eclipse Platform Plug-in Developer Guide.

Interface `IPageLayout`
Description: A page layout defines the initial layout for a perspective within a page in a workbench window... View placeholders may also have a secondary id. ... For example, the placeholder "someView:*" will match any occurrence of the view that has primary id "someView" and that also has some non-null secondary id. Note that this placeholder will not match the view if it has no secondary id ...

Figure 5: API description for `IPageLayout`.

throughout the user guide, developer guide, and API documents, the words *view* and *placeholder* frequently appear in the same context.

The distributional hypothesis [12] has provided the basis for a number of methods that use word co-occurrences in order to create vector representations of words (i.e. *word embeddings*), such that words with similar meaning have similar vectors. Inspired by the success of unsupervised learning of word representations in natural language processing, we propose a general method in which natural language statements and code snippets are projected as meaning vectors in a low-dimensional shared representation space. When applied on contexts such as the ones shown in Figures 3 to 5, the method is expected to automatically determine that the keywords *view* and *placeholder* are related semantically, and therefore the corresponding bug report and source code should have a non-zero semantic similarity.

While word embeddings have been shown to help in various NLP tasks [3, 8], to the best of our knowledge they have not been used to support text retrieval in software engineering. A number of recent approaches explore the mining of semantically related software terms. Tian et al. [41, 42] introduce SEWordSim, a software-specific word similarity database trained on Stack Overflow questions and answers. Howard et al. [13] and Yang et al. [45] infer semantically related words from code. Wang et al. [44] mine word similarities from tags in FreeCode. However, these previous proposals for word similarities do not extend them to computing similarities between documents. In this paper, we incorporate word embeddings into a modified version of the text-to-text text similarity introduced by Mihalcea et al. [28], such that word embeddings can be used to calculate the semantic similarity between SE queries and their candidate answers.

The main contributions of this paper include: an adaptation of the Skip-gram model [30] to the task of learning word embeddings for text and code in a low-dimensional shared vector space; a method for estimating the semantic similarity between queries and documents that utilizes word embeddings; an evaluation of the proposed semantic similarity features that shows their utility in two IR tasks in SE: a previously explored bug localization task and a newly defined task of linking API documents to Java questions posted on Stack Overflow.

The rest of this paper is organized as follows. Section 2 introduces the word embeddings idea, with a subsection detailing the Skip-gram model for learning word embeddings. Section 3 introduces techniques to adapt the Skip-gram model for training word embeddings on software documents that contains both free-text and code. Section 4 introduces two similarity features that are aimed at capturing the semantic similarity between documents using word embeddings. Section 5 describes the general ranking model used to implement the two IR systems evaluated in this paper. Section 6 presents a comprehensive evaluation of the impact that the newly proposed features have on the ranking performance. Section 8 discusses related work, followed by conclusions in Section 10.

2. WORD EMBEDDINGS

Harris' [12] distributional hypothesis, which states that words in the same context tends to have similar meanings, has given rise to many *distributional semantic models* (DSMs) in which individual words are no longer treated as unique symbols, but represented as d-dimensional vectors of real numbers that capture their contextual semantic meanings,

such that similar words have similar vector representations.

Traditional DSMs create word vectors from a word-context matrix M, where each row corresponds to a word w_i, each column to a context c_j that contains w_i, and each cell M_{ij} to the co-occurrence counts (e.g. the occurrences of w_i in c_j). For example, Pantel and Lin [33] compute M_{ij} as the pointwise mutual information (PMI) between w_i and c_j. Landauer and Dumais [19] introduced Latent Semantic Analysis (LSA), which applies truncated Singular Value Decomposition (SVD) to reduce M to a low-dimensional latent space. These models are referred to as count-based models.

Recently, a set of neural-network-based approaches [4, 7, 30] were proposed to represent each word with a low-dimensional vector called "neural embedding" or "word embedding" [21]. Unlike the traditional DSMs that initialize vectors with co-occurrence counts, these neural language models directly learn the vectors to optimally predict the context, and are called predictive models. Such models were successfully applied in a variety of NLP tasks [3, 8, 31]. Among these models, Mikolov's Skip-gram model [30] is popular for its simplicity and efficiency during training. The Skip-gram model was also shown to significantly outperforms LSA and other traditional count-based approaches in a set of tasks in IR and NLP [3, 31].

2.1 Learning Word Embeddings

Learning word embeddings refers to finding vector representations of words such that words that are similar in meaning are associated with similar vector embeddings, where the similarity between vectors is usually defined as cosine similarity. To learn embeddings for natural language words and code tokens, we use the unsupervised Skip-gram model of Mikolov et al. [30]. The Skip-gram model learns vector representations of words that are useful for predicting the surrounding words in a sentence. Figure 6 illustrates the training procedure employed by the Skip-gram model when it reaches the current word w_t = *numbers*.

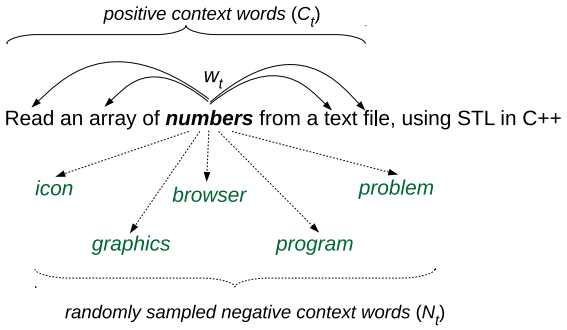

Figure 6: Positive and negative training examples in the Skip-gram model.

The vector representation $\mathbf{w}_t$ is used as the parameter vector of a binary logistic regression model (Equation 1) that takes an arbitrary word w_k as input and is trained to predict a probability of 1 if the w_k appears in the context of w_t and 0 otherwise.

$$P(w_k \in C_t | w_t) = \sigma(\mathbf{w}_t^T \mathbf{w}_k) = (1 + exp(-\mathbf{w}_t^T \mathbf{w}_k))^{-1} \quad (1)$$

Thus, given an arbitrary word w_k, its vector representation $\mathbf{w}_k$ is used as a feature vector in the logistic regression model parameterized by $\mathbf{w}_t$. If the word w_k is in the context of w_t, it is considered to be a positive example (w_+). Any other word can serve as a negative example (w_-). The context C_t is usually defined as a fixed size window centered at the current word w_t. The set of (noisy) negative examples N_t is constructed by randomly sampling from the domain vocabulary a fixed number of words, for each word in the context C_t. When trained on a sequence of T words, the Skip-gram model uses stochastic gradient descent to minimize the negative of the log-likelihood objective $J(\mathbf{w})$ shown in Equation 2.

$$J(\mathbf{w}) = \sum_{t=1}^{T} \sum_{w_+ \in C_t} (\log \sigma(\mathbf{w}_t^T \mathbf{w}_+) + \sum_{w_- \in N_t} \log \sigma(-\mathbf{w}_t^T \mathbf{w}_-)) \quad (2)$$

When applying the Skip-gram model for learning embeddings of natural language words and source code tokens, we considered the following two scenarios:

1. **One-vocabulary setting:** A single vocabulary is created to contain both words and tokens. This means that the natural language word *clear* referring to an adjective and the source code token *clear* referring to a method name would be represented with the same vocabulary entry.

2. **Two-vocabulary setting:** Two vocabularies are created. One is used for words appearing in the natural language text and the other is used for tokens appearing in the code. This means that the natural language word *clear* referring to an adjective and the source code token `clear` referring to a method name will belong to different vocabularies.

One difference between the one-vocabulary setting and the two-vocabulary setting is that the former uses the context of the method name *clear* to train the word embedding for the adjective *clear*, while the second one does not.

3. LEARNING WORD EMBEDDINGS ON SOFTWARE DOCUMENTS

Tutorials, API documents, and bug reports often contain sentences that mix natural language with code. Figure 7 shows sentences extracted from an Eclipse tutorial[5] on using OpenGL with SWT. The first sentence creates a context for

A context must be created with a `Drawable`, usually an SWT `Canvas`, on which OpenGL renders its scenes.
The application uses `GLScene`, which is a utility class for displaying OpenGL scenes. The `GLScene` class is similar to SWT's `Canvas`.
`GLScene` uses the entire area of the canvas for drawing. In the constructor, a new SWT `Canvas` is created. This is the canvas that is associated with a `GLContext` instance.

Figure 7: Tutorial sentences mixing natural language with code.

[5]http://www.eclipse.org/articles/Article-SWT-OpenGL/opengl.html

the natural language word "render", whereas the next 2 sentences create a context for the code token `GLScene`. The two contexts have a significant number of words/tokens in common: C(render) ∩ C(`GLScene`) = {SWT, `Canvas`, OpenGL, scene}. When run over a large number of sentences that contain similar contexts for the two words "render" and `GLScene`, the Skip-gram model will automatically create vector representations for the two words that are similar in the same, shared vector space.

The number of sentences that mix natural language words with source code tokens is often insufficient for training good vector representations in the Skip-gram model. Since the necessary size of the corpus is proportional with the size of the vocabulary, we reduce the natural language vocabulary by pre-processing the text with the Porter stemmer. This effectively collapses derivationally related words such as *draw, draws, drawing, drawable* into the same vocabulary entry *draw*. Furthermore, using the same corpus we increase the number of training examples for the Skip-gram model as follows:

1. Whenever a code token has a name that matches a natural language word, use both the token and the word to create training examples for the logistic regression models (Section 3.1).

2. Apply the logistic regression models on pairs of text and code that are known to have the same meaning or related meanings, such as class-description and method-description in API documentations (Section 3.2).

3.1 Heuristic Mapping of Tokens to Words

So far, under the two vocabularies setting, the word "canvas" and the token `Canvas` belong to different vocabularies and therefore are associated different vector representations. However, when naming code artifacts such as classes, methods, or variables, programmers tend to use natural language words that describe the meaning of the artifact. Furthermore, when referring to the artifact in natural language, programmers often use the natural language word instead of the code token. For example, the third section of Figure 7 mentions the creation of an object of class `Canvas`, which is later referred to using the natural language word "canvas". We exploit this naming tendencies and create additional training examples for the Skip-gram model by instructing the model to also train for the natural language word "canvas" whenever it sees an occurrence of the code token `Canvas`. This is done for each code token whose string representation is identical with the string representation of a word in the natural language vocabulary. This rule is further generalized to composite token names such as `WorkbenchWindow` by first splitting them into atomic names based on capitalization patterns, and then checking to see if the atomic code names match words in the natural language vocabulary. For example, `WorkbenchWindow` will generate the natural language words "workbench" and "window", whereas `GLScene` will generate the words "GL" and "scene" ("GL" is considered a natural language word because it appears in natural language sentences such as "Each GL command consists of the library prefix, followed by the command name"). Note that the application of this heuristic rule implicitly leads to richer contexts. In the example from Figure 7, the code token `Drawable` from the context of "render" is mapped to the natural language word "drawable", which has the same stem as the word "draw" that appears in the context of `GLScene`.

The `GLScene` class is similar to SWT's `Canvas`. However, rather than using a `GC` to draw on it, its content is rendered by OpenGL commands. This is achieved by associating a `GLContext` with an SWT `Canvas` and making it the current context whenever a scene is rendered by the commands defined in the `drawScene` method.

```
1 public class GLScene {
2     private GLContext context;
3     private Canvas canvas;
4
5     public GLScene(Composite parent) {
6         this.canvas = new Canvas(parent, SWT.NONE);
7         this.canvas.addControlListener(new ControlAdapter() {
8             public void controlResized(ControlEvent e) {
9                 resizeScene();
```

Figure 8: Example of semantically related text and code, from the same tutorial.

```
void connect(IStreamsProxy streamsProxy)
```

Connects this console to the given streams proxy. This associates the standard in, out, and error streams with the console. Keyboard input will be written to the given proxy.

Figure 9: Example of semantically related text and code, from API documents.

3.2 Semantic Pairings between Text and Code

Source code is often paired with natural language statements that describe its behavior. Such semantic pairings between text and code can be automatically extracted from tutorial examples (Figure 8) and API documents (Figure 9). To force the two meaning representations (the bag-of-word-embeddings and the bag-of-token-embeddings) to be similar, we augment the Skip-gram model to predict all code tokens from each text word, and all text words from each code token, as shown in Figure 10. This effectively pushes the embeddings for the text words and the code tokens towards each other: if w_t is a word embedding and w_k is a token embedding, both with the same norm, the logistic sigmoid in Equation 1 is maximized when $w_k = w_t$.

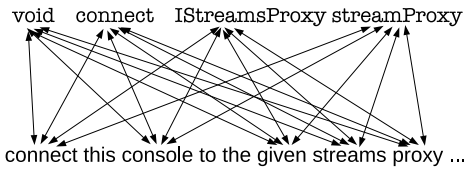

Figure 10: Positive pairs generated from semantically related text and code.

4. FROM WORD EMBEDDINGS TO DOCUMENT SIMILARITIES

Given two words w_t and w_u, we define their semantic similarity as the *cosine similarity* between their learned word embeddings:

$$sim(w_t, w_u) = cos(\mathbf{w}_t, \mathbf{w}_u) = \frac{\mathbf{w}_t^T \mathbf{w}_u}{\|\mathbf{w}_t\| \|\mathbf{w}_u\|} \quad (3)$$

This is simply the inner product of the two vectors, normalized by their Euclidean norm. To compute the similarity

between two bags-of-words T (e.g. natural language text) and S (e.g. source code), we modified the text-to-text similarity measure introduced by Mihalcea et al. [28]. According to [28], the similarity between a word w and a bag of words T is computed as the maximum similarity between w and any word w' in T:

$$sim(w, T) = \max_{w' \in T} sim(w, w') \tag{4}$$

An asymmetric similarity $sim(T \rightarrow S)$ is then computed as a normalized, *idf*-weighted sum of similarities between words in T and the entire bag-of-words in S:

$$sim(T \rightarrow S) = \frac{\sum_{w \in T} sim(w, S) * idf(w)}{\sum_{w \in T} idf(w)} \tag{5}$$

The asymmetric similarity $sim(S \rightarrow T)$ is computed analogously, by swapping S and T in the formula above. Finally, the symmetric similarity $sim(T, S)$ between two bags-of-words T and S is defined in [28] as the sum of the two asymmetric similarities:

$$sim(T, S) = sim(T \rightarrow S) + sim(S \rightarrow T) \tag{6}$$

To keep the system self-contained, we could compute the *idf* weights from the collection of documents T in the dataset. For example, in the bug localization task, the *idf* weights would be computed from the collection of source code files. However, this is bound to result in small document frequencies for many English words, which in turn make the *idf* estimates unreliable. To avoid this problem, we eliminated the *idf* weighting from the asymmetric similarities. Furthermore, we simplified the asymmetric similarity formula by ignoring words that had a zero similarity with the target document, i.e. words that either do not have a word embedding, or that do not appear in the target document. Thus, if we define $P(T \rightarrow S) = \{w \in T | sim(w, S) \neq 0\}$ to be the set of words in T that have non-zero (positive) similarity with S, the modified asymmetric similarity between documents T and S is computed as follows:

$$sim(T \rightarrow S) = \frac{\sum_{w \in P(T \rightarrow S)} sim(w, S)}{|P(T \rightarrow S)|} \tag{7}$$

The asymmetric similarity $sim(S \rightarrow T)$ is computed analogously, by swapping S and T.

5. FROM DOCUMENT SIMILARITIES TO RANKING SCORES

The information retrieval systems evaluated in this paper take as input a query document T expressing a user information need and rank all documents S from a large corpus of documents, such that the documents ranked at the top are more likely to answer the information need (i.e. relevant documents). This is usually implemented through a ranking model that computes a ranking score $f(T, S)$ for any (query, candidate answer) pair. In our work, we consider ranking functions that are defined as a weighted sum of K features, where each feature $\phi_k(T, S)$ captures a predefined relationship between the query document T and the candidate answer S:

$$f(T, S) = \mathbf{w}^T \Phi(T, S) = \sum_{k=1}^{K} w_k * \phi_k(T, S) \tag{8}$$

The feature weights w_k will be trained on a dataset of automatically acquired ranking constraints, using a learning-to-rank technique. In this setting, evaluating the impact of the new document-to-document similarities based on word embeddings can be done simply by adding the asymmetric similarities $sim(T \rightarrow S)$ and $sim(S \rightarrow T)$ as new features in an existing ranking model.

6. EVALUATION OF WORD EMBEDDINGS FOR BUG LOCALIZATION

In this section, we describe an extensive set of experiments that are intended to determine the utility of the new document similarity measures based on word embeddings in the context of bug localization. This is an information retrieval task in which queries are bug reports and the system is trained to identify relevant, buggy files.

6.1 Text Pre-processing

There are three types of text documents used in the experimental evaluations in this section: 1) the Eclipse API reference, developer guides, Java API reference, and Java tutorials that are used to train the word embeddings; 2) the bug reports; and 3) the source code files. When creating bag-of-words for these documents, we use the same pre-processing steps on all of them: we remove punctuation and numerical numbers, then split the text by whitespace.

The tokenization however is done differently for each category of documents. In the one-vocabulary setting, compound words in the bug reports and the source code files are split based on capital letters. For example, "WorkbenchWindow" is split into "Workbench" and "Window", while its original form is also reserved. We then apply the Porter stemmer on all words/tokens.

In the two-vocabulary setting, a code token such as a method name "*clear* is marked as "@clear@" so that it can be distinguished from the adjective "clear". Then we stem only the natural language words. We also split compound natural language words. In order to separate code tokens from natural language words in the training corpus, we wrote a dedicated HTML parser to recognize and mark the code tokens. For bug reports, we mark words that are not in an English dictionary as code tokens. For source code files, all tokens except those in the comments are marked as code tokens. Inside the comments, words that are not in an English dictionary are also marked as code tokens.

6.2 Corpus for Training Word Embeddings

To train the shared embeddings, we created a corpus from documents in the following Eclipse repositories: the Platform API Reference, the JDT API Reference, the Birt API Reference, the Java SE 7 API Reference, the Java tutorials, the Platform Plug-in Developer Guide, the Workbench User Guide, the Plug-in Development Environment Guide, and the JDT Plug-in Developer Guide. The number of documents and words/tokens in each repository are shown in Table 2. All documents are downloaded from their official web-

Table 1: Benchmark Projects: *Eclipse* refers to Eclipse Platform UI.*

Project	Time Range	# of bug reports used for testing	# of bug reports used for training	# of bug reports used for tuning	total
Birt	2005-06-14 – 2013-12-19	583	500	1,500	2,583
Eclipse*	2001-10-10 – 2014-01-17	1,656	500	1,500	3,656
JDT	2001-10-10 – 2014-01-14	632	500	1,500	2,632
SWT	2002-02-19 – 2014-01-17	817	500	1,500	2,817

site[6][7]. Code tokens in these documents are usually placed between special HTML tags such as ⟨*code*⟩ or emphasized with different fonts.

Table 2: Documents for training word embeddings.

Data sources	Documents	Words/Tokens
Platform API Reference	3,731	1,406,768
JDT API Reference	785	390,013
Birt API Reference	1,428	405,910
Java SE 7 API Reference	4,024	2,840,492
The Java Tutorials	1,282	1,024,358
Platform Plug-in Developer Guide	343	182,831
Workbench User Guide	426	120,734
Plug-in Development Environment Guide	269	90,356
JDT Plug-in Developer Guide	164	64,980
Total	12,452	6,526,442

Table 3: The vocabulary size.

Word embeddings trained on:	Vocabulary size
one-vocabulary setting	21,848
two-vocabulary setting	25,676

Table 4: Number of word pairs.

Approach	# of word pairs
One-vocabulary embeddings	238,612,932
Two-vocabulary embeddings	329,615,650
SEWordSim [42]	5,636,534
SWordNet [45]	1,382,246

To learn the shared embeddings, we used the Skip-gram model, modified such that it works in the training scenarios described in Section 3. Table 3 shows the number of words in each vocabulary setting. Table 4 compares the number of word pairs used to train word embeddings in the one- and two-vocabulary settings with the number of word pairs used in two related approaches. Thus, when word embeddings are trained on the one-vocabulary setting, the vocabulary size is 21,848, which leads to 238,612,932 word pairs during training. This number is over 40 times the number of word pairs in SEWordSim [42], and is more than 172 times the number of word pairs in SWordNet [45].

6.3 Benchmark Datasets

We perform evaluations on the fined-grained benchmark dataset from [46]. Specifically, we use four open-source Java projects: Birt[8], Eclipse Platform UI[9], JDT[10], and SWT[11]. For each of the 10,000 bug reports in this dataset, we *check-out* a before-fixed version of the source code, within which we rank all the source code files for the specific bug report.

Since the training corpus for word embeddings (shown in Table 2) contains only Java SE 7 documents, for testing we use only bug reports that were created for Eclipse versions starting with 3.8, which is when Eclipse started to add Java SE 7 support. The Birt, JDT, and SWT projects are all Eclipse Foundation projects, and also support Java SE 7 after the Eclipse 3.8 release. Overall, we collect for testing 583, 1656, 632, and 817 bug reports from Birt, Eclipse Platform UI, JDT, and SWT, respectively. Older bug reports that were reported for versions before release 3.8 are used for training and tuning the learning-to-rank systems.

Table 1 shows the number of bug reports from each project used in the evaluation. The methodology used to collect the bug reports is discussed at length in [46]. Here we split the bug reports into a testing set, a training set, and a tuning set. Taking Eclipse Platform UI for example, the newest 1,656 bug reports, which were reported starting with Eclipse 3.8, are used for testing. The older 500 bug reports in the training set are used for learning the weight parameters of the ranking function in Equation 8, using the SVM^{rank} package [14, 15]. The oldest 1,500 bug reports are used for tuning the hyper-parameters of the Skip-gram model and the SVM^{rank} model, by repeatedly training on 500 and testing on 1000 bug reports. To summarize, we tune the hyper-parameters of the Skip-gram model and the SVM^{rank} model on the tuning dataset, then train the weight vector used in the ranking function on the training dataset, and finally test and report the ranking performance on the testing dataset. After tuning, the Skip-gram model was train to learn embeddings of size 100, with a context window of size 10, a minimal word count of 5, and a negative sampling of 25 words.

6.4 Results and Analysis

We ran extensive experiments for the bug localization task, in order to answer the following research questions:

RQ1: Do word embeddings help improve the ranking performance, when added to an existing strong baseline?

RQ2: Do word embeddings trained on different corpora change the ranking performance?

RQ3: Do the word embedding training heuristics improve the ranking performance, when added to the vanilla Skip-gram model?

RQ4: Do the modified text similarity functions improve the ranking performance, when compared with the original similarity function in [28]?

We use the Mean Average Precision (MAP) [23], which is the mean of the average precision values for all queries, and

[6]http://docs.oracle.com/javase/7/docs
[7]http://www.eclipse.org/documentation
[8]https://www.eclipse.org/birt/
[9]http://projects.eclipse.org/projects/eclipse.platform.ui
[10]http://www.eclipse.org/jdt/
[11]http://www.eclipse.org/swt/

Table 5: MAP and MRR for the 5 ranking systems.

Project	Metric	LR+WE[1] ϕ_1-ϕ_8	LR+WE[2] ϕ_1-ϕ_8	LR ϕ_1-ϕ_6	WE[1] ϕ_7-ϕ_8	WE[2] ϕ_7-ϕ_8
Eclipse Platform UI	MAP	0.40	0.40	0.37	0.26	0.26
	MRR	0.46	0.46	0.44	0.31	0.31
JDT	MAP	0.42	0.42	0.35	0.22	0.23
	MRR	0.51	0.52	0.43	0.27	0.29
SWT	MAP	0.38	0.38	0.36	0.25	0.25
	MRR	0.45	0.45	0.43	0.30	0.30
Birt	MAP	0.21	0.21	0.19	0.13	0.13
	MRR	0.27	0.27	0.24	0.17	0.17

the Mean Reciprocal Rank (MRR) [43], which is the harmonic mean of ranks of the first relevant documents, as the evaluation metrics. MAP and MRR are standard evaluation metrics in IR, and were used previously in related work on bug localization [38, 46, 47].

6.4.1 RQ1: *Do word embeddings help improve the ranking performance?*

The results shown in Table 5 compare the **LR** system introduced in [46] with a number of systems that use word embeddings in the one- and two-vocabulary settings, as follows: **LR+WE**^{1} refers to combining the one-vocabulary word-embedding-based features with the six features of the LR system from [46], **LR+WE**^{2} refers to combining the two-vocabulary word-embedding-based features with the LR system, **WE**^{1} refers to using only the one-vocabulary word-embedding-based features, and **WE**^{2} refers to using only the two-vocabulary word-embedding-based features. The parameter vector of each ranking system is learned automatically. The results show that the new word-embedding-based similarity features, when used as additional features, improve the performance of the **LR** system. The results of both **LR+WE**^{1} and **LR+WE**^{2} show that the new features help achieve 8.1%, 20%, 5.6%, and 16.7% relative improvements in terms of MAP over the original **LR** approach, for Eclipse Platform UI, JDT, SWT, and Birt respectively. In [46], **LR** was reported to outperform other state-of-the-art bug localization models such as the VSM-based BugLocator from Zhou et al. [47] and the LDA-based BugScout from Nguyen et al. [32].

Another observation is that using word embeddings trained on one-vocabulary and using word embeddings trained on two-vocabulary achieve almost the same results. By looking at a sample of API documents and code, we discovered that class names, method names, and variable names are used with a consistent meaning throughout. For example, developers use *Window* to name a class that is used to create a window instance, and use *open* to name a method that performs an open action. Therefore, we believe the two-vocabulary setting will be more useful when word embeddings are trained on both software engineering (SE) and natural language (NL) corpora (e.g. Wikipedia), especially in situations in which a word has NL meanings that do not align well with its SE meanings. For example, since *eclipse* is used in NL mostly with the astronomical sense, it makes sense for *eclipse* to be semantically more similar with *light* than *ide*. However, in SE, we want *eclipse* to be more similar to *ide* and *platform* than to *total*, *color*, or *light*. By training separate embeddings for *eclipse* in NL contexts (i.e. *eclipse_NL*) vs. *eclipse* in SE contexts (i.e. *eclipse_SE*), the expectation is that, in an SE setting, the *eclipse_SE* embedding would be more similar with the *ide_SE* embedding than the *total_SE* or *color_SE* embeddings.

Table 6: Results on easy (T1) vs. difficult (T2) bug reports, together with # of bug reports (size) and average # of relevant files per bug report (avg).

		T1		T2	
		LR+WE[1]	LR	LR+WE[1]	LR
Eclipse	Size/Avg	322/2.11		1,334/2.89	
	MAP	0.80	0.78	0.30	0.27
	MRR	0.89	0.87	0.36	0.33
JDT	Size/Avg	84/2.60		548/2.74	
	MAP	0.79	0.75	0.36	0.29
	MRR	0.90	0.87	0.45	0.37
SWT	Size/Avg	376/2.35		441/2.57	
	MAP	0.57	0.55	0.22	0.21
	MRR	0.66	0.65	0.27	0.26
Birt	Size/Avg	27/2.48		556/2.24	
	MAP	0.48	0.54	0.20	0.17
	MRR	0.62	0.69	0.25	0.22

Kochhar et al. [17] reported from an empirical study that the localized bug reports, which explicitly mention the relevant file names, "*statistically significantly and substantially*" impact the bug localization results. They suggested that there is no need to run automatic bug localization techniques on these bug reports. Therefore, we separate the testing bug reports for each project into two subsets T1 (easy) and T2 (difficult). Bug reports in T1 mention either the relevant file names or their top-level public class names, whereas T2 contains the other bug reports. Note that, although bug reports in T1 make it easy for the programmer to find a relevant buggy file, there may be other relevant files associated with the same bug report that could be more difficult to identify, as shown in the statistics from Table 6.

Table 6 shows the MAP and MRR results on T1 and T2. Because **LR+WE**^{1} and **LR+WE**^{2} are comparable on the test bug reports, here we compare only **LR+WE**^{1} with **LR**. The results show that both **LR+WE**^{1} and **LR** achieve much better performance on bug reports in T1 than T2 for all projects. This confirms the conclusions of the empirical study from Kochhar et al. [17]. The results in Table 6 also show that overall using word embeddings helps on both T1 and T2. One exception is Birt, where the use of word embeddings hurts performance on the 27 easy bugs in T1, a result that deserves further analysis in future work.

To summarize, we showed that using word embeddings to create additional semantic similarity features helps improve the ranking performance of a state-of-the-art approach to bug localization. However, separating the code tokens from the natural language words in two vocabularies when training word embeddings on the SE corpus did not improve the performance. In future work, we plan to investigate the utility of the two-vocabulary setting when training with both SE and NL corpora.

6.4.2 RQ2: *Do word embeddings trained on a different corpora change the ranking performance?*

To test the impact of the training corpus, we train word embeddings in the one-vocabulary setting using the Wiki data dumps[12], and redo the ranking experiment. The ad-

[12]https://dumps.wikimedia.org/enwiki/

Table 7: The size of the different corpora.

Corpus	Vocabulary	Words/Tokens
Eclipse and Java	21,848	6,526,442
Wiki	2,098,556	3,581,771,341

Table 8: Comparison of the LR+WE^1 results when using word embeddings trained on different corpora.

Corpus	Metric Metric	Eclipse Platform UI	JDT	SWT	Birt
Eclipse and Java documents	MAP	0.40	0.42	0.38	0.21
	MRR	0.46	0.51	0.45	0.27
Wiki	MAP	0.40	0.41	0.38	0.21
	MRR	0.46	0.51	0.45	0.27

vantage of using the Wiki corpus is its large size for training. Table 7 shows the size of the Wiki corpus. The number of words/tokens in the Wiki corpus is 548 times the number in our corpus, while its vocabulary size is 96 times the vocabulary size of our corpus. Theoretically, the larger the size of the training corpus the better the word embeddings. On the other hand, the advantage of the smaller training corpus in Table 2 is that its vocabulary is close to the vocabulary used in the queries (bug reports) and the documents (source code files).

Table 8 shows the ranking performance by using the Wiki embeddings. Results show that the project specific embeddings achieve almost the same MAP and MRR for all projects as the Wiki embeddings. We believe one reason for the good performance of the Wiki embeddings is the preprocessing decision to split compound words such as `WorkbenchWindow` that do not appear in the Wiki vocabulary into their components words `Workbench` and `Window`, which belong to the Wiki vocabulary. Correspondingly, Table 9 below shows the results of evaluating just the word-embeddings features (**WE**^{1}) on the Eclipse project with the two types of embeddings, with and without splitting compound words. As expected, the project-specific embeddings have better performance than the Wiki-trained embeddings when compound words are not split; the comparison is reversed when splitting is used. Overall, each corpus has its own advantages: while the embeddings trained on the project-specific corpus may better capture specific SE meanings, the embeddings trained on Wikipedia may benefit from the substantially larger amount of training examples. Given the complementary advantages, in future work we plan to investigate training strategies that exploit both types of corpora.

Table 9: Project-specific vs. Wikipedia embeddings performance of WE^1 features, with and without splitting compound words.

Project	Metric	No Split	Split
Eclipse/Java	MAP	0.254	0.260
	MRR	0.307	0.310
Wikipedia	MAP	0.248	0.288
	MRR	0.300	0.346

6.4.3 RQ3: *Do the word embedding training heuristics improve the ranking performance?*

Table 10 shows the results of using the original Skip-gram model without applying the heuristic techniques discussed in Sections 3.1 and 3.2. It shows that both the enhanced and the original Skip-gram model achieve the same results most of the time. These results appear to indicate that increasing the number of training pairs for word embeddings will not lead to further improvements in ranking performance, which is compatible with the results of using the Wiki corpus vs. the much smaller project-specific corpora.

Table 10: LR+WE^1 results obtained using the enhanced vs. the original Skip-gram model.

Project	Metric	LR ϕ_1-ϕ_8	Enhanced Skip-gram ϕ_1-ϕ_6	Original Skip-gram ϕ_1-ϕ_8
Eclipse Platform UI	MAP	0.37	0.40	0.40
	MRR	0.44	0.46	0.46
JDT	MAP	0.35	0.42	0.42
	MRR	0.43	0.51	0.51
SWT	MAP	0.36	0.38	0.37
	MRR	0.43	0.45	0.44
Birt	MAP	0.19	0.21	0.21
	MRR	0.24	0.27	0.27

6.4.4 RQ4: *Do the modified text similarity functions improve the ranking performance?*

Table 11 below compares the new text similarity functions shown in Equation 7 with the original text similarity function from Mihalcea et al. [28], shown in Equation 6. In $\mathbf{WE}^1_{ori}$, the new features ϕ_7 and ϕ_8 are calculated using the one-vocabulary word embeddings and the original *idf*-weighted text similarity function. The results of **LR+WE**^{1} and **LR** are copied from Table 5, for which ϕ_7 and ϕ_8 are calculated using the new text similarity functions.

Table 11: Comparison between the new text similarity function (LR+WE^1) and the original similarity function (LR+WE$^1_{ori}$).

Project	Metric	LR ϕ_1-ϕ_8	LR+WE^1 ϕ_1-ϕ_6	LR+WE$^1_{ori}$ ϕ_7-ϕ_8
Eclipse Platform UI	MAP	0.37	0.40	0.37
	MRR	0.44	0.46	0.43
JDT	MAP	0.35	0.42	0.36
	MRR	0.43	0.51	0.45
SWT	MAP	0.36	0.38	0.37
	MRR	0.43	0.45	0.44
Birt	MAP	0.19	0.21	0.20
	MRR	0.24	0.27	0.25

Results show that the new text similarity features lead to better performance than using the original text similarity function. The new features obtain a 20% relative improvement in terms of MAP over the **LR** approach, while features calculated based on the original text similarity function achieve only a 3% relative improvement.

7. EVALUATION OF WORD EMBEDDINGS FOR API RECOMMENDATION

To assess the generality of using document similarities based on word embeddings for information retrieval in software engineering, we evaluate the new similarity functions on the problem of linking API documents to Java questions posted on the community question answering (cQA) website Stack Overflow (SO). The SO website enables users to ask

and answer computer programming questions, and also to vote on the quality of questions and answers posted on the website. In the Question-to-API (Q2API) linking task, the aim is to build a system that takes as input a user's question in order to identify API documents that have a non-trivial semantic overlap with the (as yet unknown) correct answer. We see such a system as being especially useful when users ask new questions, for which they would have to wait until other users post their answers. Recommending relevant API documents to the user may help the user find the answer on their own, possibly even before the correct answer is posted on the website. To the best of our knowledge, the Q2API task for cQA websites has not been addressed before.

In order to create a benchmark dataset, we first extracted all questions that were tagged with the keyword 'java', using the datadump archive available on the Stack Exchange website. Of the 1,493,883 extracted questions, we used a script to automatically select only the questions satisfying the following criteria:

1. The question score is larger than 20, which means that more than 20 people have voted this question as "useful".

2. The question has answers of which one was checked as the "correct" answer by the user who asked the question.

3. The "correct" answer has a score that is larger than 10, which means that more than 10 people gave a positive vote to this answer.

4. The "correct" answer contains at least one link to an API document in the official Java SE API online reference (versions 6 or 7).

This resulted in a set of high quality 604 questions, whose correct answers contain links to Java API documents. We randomly selected 150 questions and asked two proficient Java programmers to label the corresponding API links as *helpful* or *not helpful*. The remaining 454 questions were used as a (noisy) training dataset. Out of the 150 randomly sampled questions, the 111 questions that were labeled by both annotators as having *helpful* API links were used for testing. The two annotators were allowed to look at the correct answer in order to determine the semantic overlap with the API document.

Although we allow API links to both versions 6 and 7, we train the word embeddings in the one-vocabulary setting, using only the Java SE 7 API documentations and tutorials. There are 5,306 documents in total, containing 3,864,850 word tokens.

We use the Vector Space Model (VSM) as the baseline ranking system. Given a question T, for each API document S we calculate the VSM similarity as feature $\phi_1(T, S)$ and the asymmetric semantic similarities that are based on word embeddings as features $\phi_2(T, S)$ and $\phi_3(T, S)$. In the VSM+WE system, the file score of each API document is calculated as the weighted sum of these three features, as shown in Equation 8. During training on the 454 questions, the objective of the learning-to-rank system is to find weights such that, for each training question, the relevant (helpful) API documents are ranked at the top. During evaluation on the 111 questions in the test dataset, we rank all the Java API documents S for each question T in descending order of their ranking score $f(T, S)$.

Table 12: Results on the Q2API task.

Approach	MAP	MRR
VSM	0.11	0.12
VSM+WE	0.35	0.39

Table 12 shows the MAP and MRR performance of the baseline VSM system that uses only the VSM similarity feature, vs. the performance of the VSM+WE system that also uses the two semantic similarity features. The results in this table indicate that the document similarity features based on word embeddings lead to substantial improvements in performance. As such, these results can serve as an additional empirical validation of the utility of word embeddings for information retrieval tasks in software engineering.

We note that these results are by no means the best results that we expect for this task, especially since the new features were added to a rather simple VSM baseline. For example, instead of treating SO questions only as bags of undifferentiated words, the questions could additionally be parsed in order to identify code tokens or code-like words that are then disambiguated and mapped to the corresponding API entities [1, 9, 40]. Given that, like VSM, these techniques are highly lexicalized, we expect their performance to improve if used in combination with additional features based on word embeddings.

8. RELATED WORK

Related work on word embedding in NLP was discussed in Section 2. In this section we discuss other methods for computing word similarities in software engineering and related approaches for bridging the lexical gap in software engineering tasks.

8.1 Word Similarities in SE

To the best of our knowledge, word embedding techniques have not been applied before to solve information retrieval tasks in SE. However, researchers [13, 44, 45] have proposed methods to infer semantically related software terms, and have built software-specific word similarity databases [41, 42].

Tian et al. [41, 42] introduce a software-specific word similarity database called SEWordSim that was trained on StackOverflow questions and answers. They represent words in a high-dimensional space in which every element within the vector representation of word w_i is the Positive Pointwise Mutual Information (PPMI) between w_i and another word w_j in the vocabulary. Because the vector space dimension equals the vocabulary size, the scalability of their vector representation is limited by the size of the vocabulary. When the size of the training corpus grows, the growing vector dimension will lead to both larger time and space complexities. Recent studies [3, 29] also showed that this kind of traditional count-based language models were outperformed by the neural-network-based low-dimensional word embedding models on a wide range of word similarity tasks.

Howard et al. [13] and Yang et al. [45] infer semantically related words directly from comment-code, comment-comment, or code-code pairs without creating the distributional vector representations. They first need to map a

line of comment (or code) to another line of comment (or code), and then infer word pairs from these line pairs. Similarly, Wang et al. [44] infer word similarities from tags in FreeCode. The main drawback of these approaches is that they rely solely on code, comments, and tags. More general free-text contents are ignored. Many semantically related words (e.g. "placeholder" and "view") are not in the source code but in the free-text contents of project documents (e.g. the Eclipse user guide, developer guide, and API document shown in Figure 3 to Figure 5). However, these types of documents are not exploited in these approaches.

More importantly, all the above approaches did not explain how word similarities can be used to estimate document similarities. They reported user studies in which human subjects were recruited to evaluate whether the word similarities are accurate. However, these subjective evaluations do not tell whether and how word similarities can be used in solving IR tasks in SE.

8.2 Bridging the Lexical Gap to Support Software Engineering Tasks

Text retrieval techniques have been shown to help in various SE tasks [11, 24]. However, the system performance is usually suboptimal due to the lexical gap between user queries and code [27]. To bridge the lexical gap, a number of approaches [2, 5, 10, 27, 39, 46] have been recently proposed that exploit information from API documentations. These approaches extract API entities referenced in code, and use the corresponding documentations to enhance the ranking results.

Specifically, McMillan et al. [27] measure the lexical similarity between the user query and API entities, then rank higher the code that uses the API entities with higher similarity scores. Bajracharya et al. [2] augment the code with tokens from other code segments that use the same API entries. Ye et al. [46] concatenate the descriptions of all API entries used in the code, and directly measure the lexical similarity between the query and the concatenated document. The main drawback of these approaches is that they consider only the API entities used in the code. The documentations of other API entities are not used. Figure 1 shows the Eclipse bug 384108. Figure 2 shows its relevant file *PartServiceImpl.java*. Figure 5 shows the description of an API entry *IPageLayout*. Although *IPageLayout* is not used in *PartServiceImpl.java*, its API descriptions contains useful information that can help bridge the lexical gap by mapping the term "view" in bug 384108 with the term "placeholder" in *PartServiceImpl.java*. Therefore, to bridge the lexical gap, we should consider not only the descriptions of the API entities used in the code but also all API documents and project documents (e.g. the user guide shown in Figure 3 and the developer guide in Figure 4) that are available.

Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA) have been used in the area of feature location and bug localization. Poshyvanyk et al. [35, 36] use LSI to reduce the dimension of the term-document matrix, represent code and queries as vectors, and estimate the similarity between code and queries using the cosine similarity between their vector representations. Similarly, Nguyen et al. [32] and Lukins et al. [22] use LDA to represent code and queries as topic distribution vectors. Rao et al. [37] compare various IR techniques on bug localization, and report that traditional IR techniques such as VSM and Unigram Model (UM) outperform the more sophisticated LSI and LDA techniques. These approaches create vector representations for documents instead of words and estimate query-code similarity based on the cosine similarity between their vectors. McMillan et al. [26] introduced a LSI-based approach for measuring program similarity, and showed that their model achieve higher precision than a LSA-based approach [16] in detecting similar applications. All these works neither measure word similarities nor try to bridge the lexical gap between code and queries.

9. FUTURE WORK

We plan to explore alternative methods for aggregating word-level similarities into a document-level similarity function, such as the Word Mover's Distance recently proposed in [18]. In parallel, we will explore methods that train document embeddings directly, such as the Paragraph Vectors of Le and Mikolov [20], and investigate their generalization from shallow bags-of-words inputs to higher level structures, such as sequences and (abstract) syntax trees.

10. CONCLUSION

We introduced a general approach to bridging the lexical gap between natural language text and source code by projecting text and code as meaning vectors into a shared representation space. In the proposed architecture, word embeddings are first trained on API documents, tutorials, and reference documents, and then aggregated in order to estimate semantic similarities between documents. Empirical evaluations show that the learned vector space embeddings lead to improvements when added to a previous state-of-the-art approach to bug localization. Furthermore, preliminary experiments on a newly defined task of linking API documents to computer programming questions show that word embeddings also improve the performance of a simple VSM baseline on this task.

11. ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful comments and suggestions. This work was partially supported by the NSF GK-12 BooKS project under grant No. 0947813.

12. REFERENCES

[1] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proc. ICSE '10*, pages 375–384, 2010.

[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. FSE '10*, pages 157–166, 2010.

[3] M. Baroni, G. Dinu, and G. Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proc. ACL '14*, pages 238–247, Baltimore, Maryland, 2014.

[4] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, Mar. 2003.

[5] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Proc. FASE '09*, pages 385–400, 2009.

[6] C. Chen and K. Zhang. Who asked what: Integrating crowdsourced FAQs into API documentation. In *Proc. ICSE '14 Companion*, pages 456–459, 2014.

[7] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proc. ICML '08*, pages 160–167, 2008.

[8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, Nov. 2011.

[9] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proc. ICSE '12*, pages 47–57, 2012.

[10] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *Proc. ICSM '13*, pages 320–329, 2013.

[11] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proc. ICSE '13*, pages 842–851, 2013.

[12] Z. Harris. Distributional structure. *Word*, 10(23):146–162, 1954.

[13] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proc. MSR '13*, pages 377–386, 2013.

[14] T. Joachims. Optimizing search engines using clickthrough data. In *Proc. KDD '02*, pages 133–142, 2002.

[15] T. Joachims. Training linear SVMs in linear time. In *Proc. KDD '06*, pages 217–226, 2006.

[16] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue. MUDABlue: an automatic categorization system for open source repositories. In *Proc. APSEC '04*, pages 184–193, 2004.

[17] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proc. ASE '14*, pages 803–814, 2014.

[18] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. From word embeddings to document distances. In *Proc. of ICML*, 2015.

[19] T. LANDAUER and S. DUMAIS. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211–240, 1997.

[20] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proc. ICML '14*, pages 1188–1196, 2014.

[21] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Proc. NIPS 27*, pages 2177–2185. 2014.

[22] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using Latent Dirichlet Allocation. *Inf. Softw. Technol.*, 52(9):972–990, Sept. 2010.

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

[24] A. Marcus and G. Antoniol. On the use of text retrieval techniques in software engineering. In *Proc. ICSE '12, Technical Briefing*, 2012.

[25] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. WCRE '04*, pages 214–223, 2004.

[26] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proc. ICSE '12*, pages 364–374, 2012.

[27] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *Trans. Softw. Eng.*, 38(5):1069–1087, Sept 2012.

[28] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proc. AAAI '06*, pages 775–780, 2006.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proc. of Workshop at ICLR '13*, 2013.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS 26*, pages 3111–3119, 2013.

[31] T. Mikolov, W.-T. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proc. NAACL-HLT-2013*, 2013.

[32] 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 *Proc. ASE '11*, pages 263–272, 2011.

[33] P. Pantel and D. Lin. Discovering word senses from text. In *Proc. KDD '02*, pages 613–619, 2002.

[34] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow. *Technical Report GIT-CS-12-05, Georgia Institute of Technology*, May 2012.

[35] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, June 2007.

[36] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and Latent Semantic Indexing for feature identification. In *Proc. ICPC '06*, pages 137–148, 2006.

[37] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proc. MSR '11*, pages 43–52, 2011.

[38] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Proc. ASE'13*, pages 345–355, 2013.

[39] J. Stylos and B. A. Myers. Mica: A Web-search tool for finding API components and examples. In *Proc. VLHCC '06*, pages 195–202, 2006.

[40] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proc. ICSE '14*, pages 643–652, 2014.

[41] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Proc. CSMR-WCRE '14*, pages 44–53, 2014.

[42] Y. Tian, D. Lo, and J. Lawall. SEWordSim: Software-specific word similarity database. In *Proc. ICSE '14 Companion*, pages 568–571, 2014.

[43] E. M. Voorhees. The TREC-8 question answering track report. In *Proc. TREC-8*, pages 77–82, 1999.

[44] S. Wang, D. Lo, and L. Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *Proc. ICSM '12*, pages 604–607, 2012.

[45] J. Yang and L. Tan. Inferring semantically related words from software context. In *Proc. MSR '12*, pages 161–170, 2012.

[46] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proc. FSE '14*, pages 689–699, 2014.

[47] 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 *Proc. ICSE '12*, pages 14–24, 2012.