DPWord2Vec: Better Representation of Design Patterns in Semantics

Dong Liu, He Jiang, *Member, IEEE,* Xiaochen Li, Zhilei Ren, Lei Qiao, and Zuohua Ding, *Member, IEEE*

Abstract—With the plain text descriptions of design patterns, developers could better learn and understand the definitions and usage scenarios of design patterns. To facilitate the automatic usage of these descriptions, e.g., recommending design patterns by free-text queries, design patterns and natural languages should be adequately associated. Existing studies usually use texts in design pattern books as the representations of design patterns to calculate similarities with the queries. However, this way is problematic. Lots of information of design patterns may be absent from design pattern books and many words would be out of vocabulary due to the content limitation of these books. To overcome these issues, a more comprehensive method should be constructed to estimate the relatedness between design patterns and natural language words. Motivated by Word2Vec, in this study, we propose DPWord2Vec that embeds design patterns and natural language words into vectors simultaneously. We first build a corpus containing more than 400 thousand documents extracted from design pattern books, Wikipedia, and Stack Overflow. Next, we redefine the concept of context window to associate design patterns with words. Then, the design pattern and word vector representations are learnt by leveraging an advanced word embedding method. The learnt design pattern and word vectors can be universally used in textual description based design pattern tasks. An evaluation shows that DPWord2Vec outperforms the baseline algorithms by 24.2%-120.9% in measuring the similarities between design patterns and words in terms of Spearman's rank correlation coefficient. Moreover, we adopt DPWord2Vec on two typical design pattern tasks. In the design pattern tag recommendation task, the DPWord2Vec-based method outperforms two state-of-the-art algorithms by 6.6% and 32.7% respectively when considering $Recall@10$. In the design pattern selection task, DPWord2Vec improves the existing methods by 6.5%-70.7% in terms of MRR.

Index Terms—Design Pattern, Word Embedding, Word2Vec, Semantic Similarity, Tag Recommendation, Design Pattern Selection.

1 Introduction

Software design patterns derive from the notion of design pattern in the area of architecture [1], aiming to document reusable experience for recurring software design problems [2]. In recent years, many studies about design patterns have been conducted [3], [4], [5]. As to the literature, there are roughly two ways to describe design patterns: the formal way and the informal way.

The formal way specifies design patterns with formally defined pattern languages. For example, the GoF (Gang-of-Four) book respectively uses UML (Unified Modeling Language) class diagram and sequence diagram to illustrate the structure and collaborations of each design pattern [2]. A number of studies are based on the formal descriptions of design patterns [4], [6], as formal specifications enhance the capabilities of machine processing [7]. However, there are some weaknesses of the formal way. Firstly, it is inconvenient to precisely specify the intent and applicability of design patterns. Secondly, building the meta-model of each design pattern is usually costly [8]. Thirdly, the formal way may lose human readability, which is critically important to the utility of design patterns [7].

D. Liu is with School of Software, Dalian University of Technology, Dalian, China. E-mail: dongliu@mail.dlut.edu.cn
H. Jiang and Z. Ren are with School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. E-mail: {jianghe,zren}@dlut.edu.cn
X. Li is with School of Software, Dalian University of Technology, Dalian, China, and University of Luxembourg, Luxembourg. E-mail: xiaochen.li@uni.lu
L. Qiao is with Beijing Institute of Control Engineering, Beijing, China. E-mail: fly2moon@aliyun.com
Z. Ding is with School of Information Sciences, Zhejiang Sci-Tech University, Hangzhou, China. E-mail: zouhuading@hotmail.com

Conversely, the informal way depicts design patterns with free text. Comparing with the formal way, it is more understandable and convenient to describe design pattern relevant artifacts in words. Thus, the informal way is a profitable supplement to the formal way. To provide tool supports for design pattern relevant tasks based on informal descriptions, the key point is to establish the semantic relationships between design patterns and natural languages, so that the retrieval or identification of design patterns can be practically realized. However, to associate design patterns with natural languages is no easy job. A design pattern name is usually a phrase, such as "factory method". An experienced developer may capture the semantics of the design pattern via the name as he/she understands the relevant background. But for the automatic tools, it is difficult to comprehend the connotations from only these several words. More information about design patterns should be provided for them to "learn" the background knowledge.

To obtain exact semantic information of design patterns, the existing studies usually take the descriptions in design pattern books as standard definitions of design patterns [8], [9], [10]. If a snippet of text is similar to the standard definition of a design pattern, then it is likely to be related to the design pattern. Hence, the relatedness between design patterns and natural languages can be estimated. However, this kind of methods is still problematic. On one hand, much information about design patterns is absent from these books. Design pattern books usually depict the mechanisms, scenarios, and specifications of design patterns [2]. As time goes by, many applications beyond the original design pattern books have been developed. For example,

the Active Record design pattern is related to the Ruby on Rails framework as Active Record provides the data model of the framework[1]. The AngularJS framework implements the Dependency Injection design pattern itself and usually accompanies by this design pattern[2]. These relationships cannot be mined from design pattern books. On the other hand, the vocabulary extracted from design pattern books is usually too small. The lengths of descriptions in design pattern books are limited and many natural language words may be out of the scope. It is difficult to handle the texts containing many out-of-vocabulary words. Therefore, the wide usage of this kind of methods is restricted.

In this study, we aim to overcome these issues by constructing a general method to estimate the relatedness between design patterns and natural language words, in order that it can be universally used in the tasks based on informal descriptions of design patterns. The "words" here refer to as both plain natural language words, such as "factory" and "method", and software specific terms, such as "angularjs". Inspired by the word embedding method [11], we propose DPWord2Vec that maps both design patterns and natural language words into one vector space. With the design pattern and word vectors, the similarity between a design pattern and a word or a document can be calculated. In this way, the relationship between natural languages and design patterns can be built. However, there are two challenges to be addressed. First, how to find a relatively large corpus about design patterns? Second, how to associate a design pattern with its relevant natural language words for vectors training?

To handle the first challenge, we build a general corpus containing 491,555 documents. The general corpus consists of two parts: the description corpus and the crowdsourced corpus. The description corpus contains relatively formal design pattern descriptions that are extracted from design pattern books and Wikipedia. The crowdsourced corpus is constructed based on a set of design pattern relevant Stack Overflow posts obtained from our previous work [12]. Then we extend the concept of context window in Word2Vec to our general corpus and define the context windows for each design pattern and each word respectively. In this way, the linkages between design patterns and words are established, that is, the design pattern context windows contain words and design patterns appear in word context windows. Hence, the second challenge can be properly addressed. Finally, the design pattern and word vector representations are learnt by leveraging an advanced word embedding method, namely GloVe [13], based on these context windows.

To clarify the quality of the learnt design pattern and word vectors, we deploy an evaluation with a dp-word (design pattern - word) similarity task. Experimental results on 2,000 manually labelled dp-word pairs show that the learnt vectors by DPWord2Vec are more effective than some widely used semantic relatedness estimation algorithms, i.e., outperform these algorithms by 24.2%-120.9% in terms of Spearman's rank correlation coefficient. To show the practicability, we depict two applications of DPWord2Vec to solve two typical design pattern tasks, i.e., design pattern tag recommendation and design pattern selection. In the first application, when recommending the top 10 design pattern tags for the posts in a software information site, the DPWord2Vec-based method outperforms two state-of-the-art tag recommendation methods by 6.6% and 32.7% respectively in terms of $Recall@10$. In the second application, the method refined by DPWord2Vec outperforms the two existing design pattern selection methods by 6.5% and 70.7% respectively when considering the mean values of MRR (Mean Reciprocal Rank) over three design pattern collections.

In this paper, we make the following contributions:

1) We propose DPWord2Vec that maps both design patterns and natural language words into vectors to support design pattern relevant tasks. To the best of our knowledge, this is the first work that establishes the universal relationship between design patterns and natural languages.
2) We evaluate DPWord2Vec on a manually labelled dp-word pair dataset to show its effectiveness in semantic relatedness estimation.
3) DPWord2Vec is applied to two design pattern relevant tasks, namely design pattern tag recommendation and design pattern selection. DPWord2Vec outperforms the state-of-the-art methods.

The rest of this paper is organized as follows. Section 2 shows the background of the study. Section 3 presents the framework of DPWord2Vec. The settings and results for evaluating DPWord2Vec are depicted in Section 4 and 5, respectively. Section 6 and 7 introduce two applications of DPWord2Vec. Section 8 discusses potential threats to validity. Some studies related to our work are outlined in Section 9. We conclude the paper in Section 10.

2 Preliminaries

Before the depiction of DPWord2Vec, we demonstrate the concept of design pattern in this study and briefly introduce the word embedding technique.

2.1 Concept of Design Pattern

Generally speaking, design patterns are proven solutions to recurring software design problems [2]. However, to the best of our knowledge, there are no formal definitions nor standard lists of design patterns. There exist numbers of design pattern collections that are published with multiple channels, such as design pattern books, academic papers, or online libraries [7]. Design patterns in different collections may be depicted in different ways, e.g., in flat text format or using UML. In this paper, we focus on the design patterns with rich textual descriptions and collect design patterns from various sources.

Similar to "design pattern", "architecture pattern" is also a means for software design. Strictly, they are not a same concept, but the boundary between them may not be unified for different design pattern collections. For example, Model View Controller is an example of architectural pattern in Wikipedia[3] but marked as a design pattern in MSDN[4].

1. https://guides.rubyonrails.org/active_record_basics.html
2. https://angular.io/guide/dependency-injection
3. https://en.wikipedia.org/wiki/Architectural_pattern
4. https://msdn.microsoft.com/en-us/library/ms978748.aspx

Therefore, instead of creating a standard subjectively, we choose not to distinguish them in our study. Once an entity is identified as a design pattern in some design pattern collections, we regard it as a design pattern.

2.2 Word Embedding

Word embedding is a set of techniques that maps words or phrases in the vocabulary to vectors of real numbers. The core part of DPWord2Vec is also word embedding, but it handles both words and design patterns. Word embedding methods focus on mapping words into a continuous vector space with a much lower dimension than the size of vocabulary and the vector representation of each word is determined by supervised learning based on the corpus [11].

To facilitate the demonstration, we explain how word embedding works with an example. Assuming there is a corpus that contains a sentence: "software design patterns encapsulate proven solutions that address recurring problems". To mine the relationships between words, the sliding context window strategy is usually used [11]. A context window contains a central word and several surrounding words which are at a distance of no more than c positions from the central word. For example, the context window with centre "patterns" and c = 2 contains the surrounding words "software", "design", "encapsulate", and "proven". Multiple local context windows are constructed as the central word slides from the beginning ("software") to the end ("problems") of the corpus.

Then the word vectors are learnt based on these local context windows. The intuition is that if two words appear frequently in the same context window then their vector representations are highly associated. For example, the objective of the Skip-gram model is to learn word vector representations that are good at predicting each surrounding word by the vector of the central word [11]. Conversely, the Continuous Bag-of-Words (CBOW) model aims to predict the central word by the concatenation or average of the vectors of the surrounding words [11]. Different from them, the GloVe model counts the number of the total co-occurrences of each pair of words through all the local context windows and predicts the co-occurrence number by the vectors of the words in the pair [13].

3 The DPWord2Vec Framework

DPWord2Vec aims to embed natural language words and design patterns into one vector space. This process can be divided into four phases (as shown in Fig. 1). At first, the corpus related to design patterns are acquired from multiple sources. Next, the documents in the corpus are preprocessed. Then, we propose a context window-based strategy to strengthen the tie between words and design patterns. At last, the word and design pattern vectors are trained based on the corpus and the context windows.

3.1 Corpus Building

To train the vectors of words and design patterns, a corpus relevant to design patterns should be built at first. Formally, we construct a general corpus C, which contains multiple

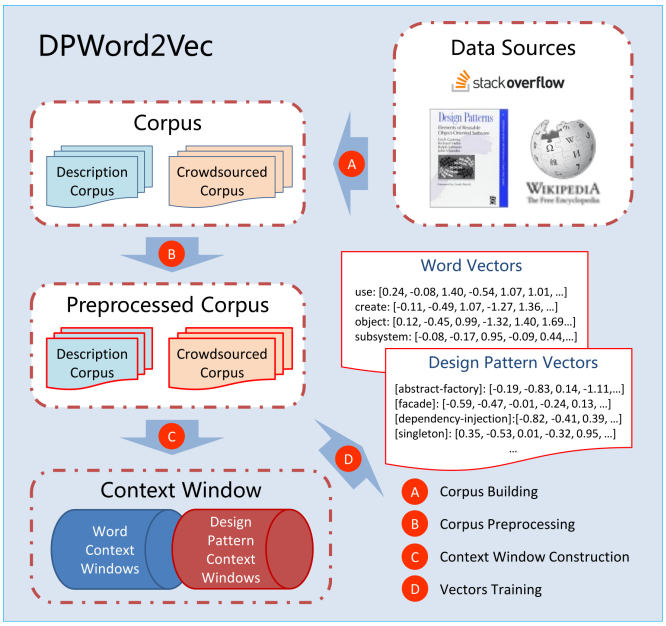


Fig. 1: The framework of DPWord2Vec.

documents. For each document doc in C, doc has two components: the token component $doc.Tokens$, a sequence of natural language words that describes some design patterns, and the design pattern component $doc.DPs$, a set of design patterns described by $doc.Tokens$. The general corpus C can be further categorized into two groups according to their sources.

Description Corpus. Documents in this corpus are extracted from design pattern books and Wikipedia. Some design pattern books catalog their own lists of design patterns. For example, GoF presents 23 design patterns with the problem definitions and design specifications [2]. A design pattern is usually described by a chapter or a section in a design pattern book. Similarly, a number of design patterns are specified by Wikipedia as entries with one page for each design pattern[5]. A chapter or section of a design pattern book, or a Wikipedia page of a design pattern forms a document doc. In this corpus, $doc.Tokens$ denotes the whole text in the chapter, section, or page, but excluding the code snippets. Meanwhile, $doc.DPs$ contains only one element, i.e., the described design pattern.

Totally, the description corpus contains 431 documents, which are associated with 13 design pattern books and 125 Wikipedia pages. Amongst the design pattern components, 349 unique design patterns are involved.

Crowdsourced Corpus. Documents in this corpus are constructed by referring to the programming forum, i.e., Stack Overflow[6]. In the previous study [12], 187,493 design pattern relevant question posts spanning from August 2008 to December 2017 are detected in Stack Overflow.

A design pattern relevant post indicates the design pattern name(s) appears at least one time in the post. However, it is not a trivial string matching task to detect the design pattern occurrences in Stack Overflow posts, as the discus-

5. https://en.wikipedia.org/wiki/Category:Software design patterns
6. https://stackoverflow.com/

sions on Stack Overflow are usually informal [14], [15] and the name of a design pattern may not be mentioned in a unique form. It is also referred to as the morphological form issue [14]. The previous study has attempted to address this issue in two aspects. On the one hand, the standard design pattern names as well as other common names are collected simultaneously from the existing design pattern collections, e.g., design pattern books, in which the other well-known names of each containing design pattern are usually presented explicitly, e.g., marked as "also known as". These names include aliases, e.g., "open implementation" is an alias for "reflection", and acronyms, e.g., "mvc" is an acronym for "model view controller". On the other hand, regular expressions are leveraged to allow some variants when searching a design pattern name in the text of the Stack Overflow posts. For example, the regular expression for "model view controller" is "model[ˆa-z]?view[ˆa-z]?controller", where "[ˆa-z]?" denotes a non-alphabetic character that matches zero or one time, so that the variants such as "model-view-controller", "model_view_controller", and "modelviewcontroller" can be involved. A manual validation on the sampled posts shows that the detection is acceptably accurate, i.e., achieves Precision value of 97.3% and Recall value of 87.8%. More details can be obtained by referring to [12].

We use these question posts to construct the crowdsourced corpus. Moreover, it is enriched by all the answer posts to these design pattern relevant question posts. A question post and each of its answer post are assigned to different documents. The relevant design pattern(s) to an answer post is as same as its question post. For a document doc in this corpus, $doc.Tokens$ denotes a content merging the title and body part of a question or answer post with code snippets discarded, and $doc.DPs$ is the set of the relevant design pattern(s) to the post.

Finally, there are 491,124 documents in this corpus and 210 unique design patterns are involved.

By merging the two corpora, we obtain a general corpus C, which contains 491,555 documents[7]. The involved design patterns are indexed and form a design pattern vocabulary, namely V_{DP}, with 372 design patterns. Although the documents in the description corpus are far less than those in the crowdsourced corpus, the description corpus is indispensable for building the design pattern vectors. On one hand, the description corpus makes it possible to build vectors for the design patterns that are rarely discussed in Stack Overflow. On the other hand, this corpus tends to provide more formal and precise depictions of design patterns than the crowdsourced corpus. We will show its significance in Section 5.1.

3.2 Corpus Preprocessing

Comparing to the general natural language documents, the amount of design pattern relevant documents tends to be quite small. Therefore, our built corpus is relatively smaller than those for training the common word vectors [11], [13]. Based on this actuality, we perform preprocessing on the token component of each document aiming to filter out the insignificant and redundant information and build a compact vocabulary.

At first, code-like tokens (e.g., function names) in a natural language sentence are split according to its camel style to ensure the semantic integrity of the sentence. With this step, on the one hand, these code-like tokens can be converted into more understandable identifiers [16] to better reflect the semantic meanings. On the other hand, the volume of the vocabulary can be reduced. Next, we tokenize and lowercase the token component of each document. Then, the less informative tokens, including English-language stop words, special tags (HTML tags in Stack Overflow posts, and reference markers in design pattern books and Wikipedia pages), and non-alphabetic characters (e.g., numbers) are removed from the text, as they are not very useful to reflect the semantic relationship between the natural language and design patterns. Moreover, each token is stemmed to its root form, e.g., "developer", "developed", and "developing" to "develop". As the words with a same root usually have similar meaning [17] and the vector representations of them are also similar in some word embedding methods [18], [19], we can simply regard them as a same word without losing much semantic information. At last, we discard the words that occur no more than five times in the corpus when constructing the vocabulary but retain them in the corpus. These words are likely to be noisy terms [20] and it is not significant to train the vectors of them.

Some of the above steps, such as camel case splitting, stop words removing, and word stemming, may be not common in word embedding methods. With abundant training corpora, vector representation of each distinct identifier in the text can be learnt. However, due to the scale of the design pattern corpus, it is reasonable to conduct these preprocessing steps to reduce the vocabulary size, i.e., the number of vectors to be learnt, to adapt to the corpus. Furthermore, the focus of this study is to build the semantic relationship between natural languages and design patterns, it is not a core concern to represent all the identifiers precisely. As a common concept in the word embedding methods, the word context will not be significantly affected by the preprocessing, since the eliminated tokens contain little semantic information and the meanings of the changed tokens are mainly retained. It is adequate to apply the word embedding methods to the preprocessed corpus.

After the preprocessing, we obtain a word vocabulary V_{Word} that contains 27,770 words.

3.3 Context Window Construction

As to the corpus we build, each document contains two parts: the natural language words and the design patterns. To train the vectors of words and design patterns together, we should combine the two parts. In standard word embedding models, words are usually associated by leveraging the sliding context window-based strategy [11]. For example, in the Skip-gram model, the vector representation of the central word is learned for predicting the other words in a context window. Similarly, the CBOW model uses the composition of the vectors of the surrounding words in a context window to predict the central word. Hence, a

7. The detailed description corpus and crowdsourced corpus, as well as the number of relevant documents to each design pattern are available via https://github.com/WoodenHeadoo/dpword2vec.

> **Dependency Injection** *is a practice where objects* are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally. This means that any object implementing the interface which is required by the object can be substituted in without changing the code, which simplifies testing, and improves decoupling.

Fig. 2: A paragraph that describes the Dependency Injection design pattern. The design pattern name is in red bold font and the words in the context window (of size five) of the name are in blue italic font.

reasonable method for associating natural language words and design patterns is to locate them in a context window.

To this end, an intuitive way is to regard design pattern names appearing in natural language text as special "words". Concretely, given a document doc in the corpus C, for the design patterns in $doc.DPs$, we detect all the occurrences of design pattern names (including aliases) in $doc.Tokens$ and replace them with predefined tokens. These predefined tokens are the "words" of design patterns and mixed with the natural language words. Then design patterns can be handled together with natural language words by the sliding context window-based strategies. However, there is a main issue for this way: design pattern names tend to appear infrequently in the text. For instance, Fig. 2 presents a paragraph in a post (#131766) of Stack Overflow. This paragraph indeed describes the Dependency Injection design pattern, but the design pattern name only appears one time at the beginning of the paragraph. When applying the sliding context window-based strategies to this paragraph, the design pattern Dependency Injection can be only associated with some words in the front but the rest are ignored.

To resolve this issue, we redefine the concept of context window by considering both natural language words and design patterns. In the new definition, the context window size is not fixed, but there is also a parameter of context window size for words as the standard models. For clarity, we name it as c.

There are two types of context windows:

Context Window for Word. For a word in a document, the context window for this word contains other words around the word with radius c and all the design patterns the document describes. Formally, for a document doc in C, let $doc.Tokens(i)$ denote the ith word of the text and $doc.Tokens.len$ denote the length of the text. The Context Window of $doc.Tokens(i)$ is defined as

$$\begin{aligned} &Context^{Word}_{doc}(i, doc.Tokens(i)) \\ =&\{doc.Tokens(j) | max\{1, i-c\} \leq j \leq \\ &min\{doc.Tokens.len, i+c\}, j \neq i\} \cup doc.DPs. \end{aligned} \tag{1}$$

Take the document in Table 1 as an example. Assuming $c = 2$, the Context Window for the sixth word "interface" contains the two words ahead of it (i.e., "facade" and "provide"), the two words behind it (i.e., "create" and "subsystem"), as well as the two design patterns mentioned in the document (i.e., "[abstract-factory]" and "[facade]").

Context Window for Design Pattern. Given a design pattern described by a document, the context window for the design pattern consists of all the words in the text and the other described design patterns. Formally, for a document doc and a design pattern $dp \in doc.DPs$, the Context Window of dp is

$$\begin{aligned} &Context^{DP}_{doc}(dp) \\ =&\{doc.Tokens(j) | 1 \leq j \leq doc.Tokens.len\} \\ &\cup (doc.DPs - \{dp\}). \end{aligned} \tag{2}$$

For example, in Table 1, the Context Window for the design pattern "[abstract-factory]" contains all the words (i.e., "abstract", "factory", ..., "class") and the other design pattern "[facade]".

According to the definitions of the two context windows, a design pattern can be associated with each word in the document that describes the design pattern. The tie between words and design patterns is strengthened. To show the effectiveness of the new definitions, we use the performance of the method that leverages design pattern name occurrences (mentioned above) for comparison in Section 5.3.

With the definitions, for any document doc in C, the context window of each word in $doc.Tokens$ and the context window of each design pattern in $doc.DPs$ are constructed.

3.4 Vectors Training

Once the context windows are clarified, the word and design pattern vectors can be generated by any sliding context window-based models. In DPWord2Vec, we choose GloVe [13] for vector generation, due to the following reasons:

1) GloVe is a state-of-the-art model that outperforms Skip-gram and CBOW on several natural language processing tasks with higher efficiency [13].
2) GloVe benefits from both global co-occurrences and local context windows. Global co-occurrences suit to present the dp-word relationships and design pattern - design pattern relationships. Meanwhile, the word - word relationships could be well handled by local context windows. Therefore, the GloVe model is suitable for this scenario.

To train the vectors with GloVe, the input of GloVe should be specified. Generally, the input of GloVe is the entries co-occurrence counts matrix X, whose element X_{ij} represents the number of times entry j occurs in the context window of entry i. In DPWord2Vec, entry j and entry i can be any word in the word vocabulary V_{Word} or any design pattern in the design pattern vocabulary V_{DP}. Therefore, in DPWord2Vec, X_{ij} is calculated respectively when entry i is a word and when entry i is a design pattern according to the two definitions of context window. Note that $X_{ij} = X_{ji}$ for any j and i according to our context window definitions, hence only half of the entries co-occurrence counts should be calculated.

Moreover, the dp-word co-occurrences are weighted. According to [21], the frequencies of words follow Zipf's law in natural language corpora. Similarly, the number of relevant posts in Stack Overflow to each design pattern exhibits a long tail behavior [12]. That means, the distribution

TABLE 1: An example for two types of Context Windows ($c = 2$)

$doc.Tokens$[a]	$doc.DPs$
[1]Abstract [2]factory ~~can be~~ [3]used(use) [4]facade ~~to~~ [5]provide ~~an~~ [6]interface ~~for~~ [7]creating(create) [8]subsystem [9]objects(object) ~~in a~~ [10]subsystem [11]independent [12]way. [13]Abstract [14]factory ~~can~~ [15]also ~~be~~ [16]used(use) ~~as an~~ [17]alternative ~~to~~ [18]facade ~~to~~ [19]hide [20]platform [21]specific [22]classes(class).	[abstract-factory], [facade]
$Context^{Word}_{doc}$(6, "interface") = {"facade", "provide", "create", "subsystem", "[abstract-factory]", "[facade]"}	
$Context^{DP}_{doc}$("[abstract-factory]") = {"abstract", "factory", ..., "class", '[facade]"}	

[a] As declared above, the stop words are eliminated from the text of the document (in strikeout fonts) and the rest of the words are stemmed to their root forms.

of words or design patterns is highly skewed. Moreover, according to the definitions, the context window of a design pattern contains all the words in the document and the design pattern is also contained in the context window of each of the words. As a result, some design patterns may appear commonly in the context windows of many words, i.e., potentially relate to many words, and vice versa. When dealing with the tasks which request to associate design patterns with words, e.g., to retrieve design patterns by keywords, we should ensure these very common design patterns not to be over weighted. Likewise, the words that are contained in the context windows of many design patterns should also be well handled. Hence, a weighting strategy is applied to diminish the effects of these common terms. Formally, if entry j is a word and entry i is a design pattern, X_{ij} is tuned by the weights of j and i. The weights are calculated just like the inverse document frequency value:

$$w_j = log(\frac{\#V_{DP}}{Occur_{DP}(j)}), w_i = log(\frac{\#V_{Word}}{Occur_{Word}(i)}), \quad (3)$$

where $Occur_{DP}(j)$ denotes the number of unique design patterns in V_{DP} that ever occur in the context window of word j and $Occur_{Word}(i)$ denotes the number of unique words in V_{Word} that ever occur in the context window of design pattern i. The weights are normalized by the average values:

$$\widetilde{w_j} = \frac{w_j}{avg\{w_{j'}|j' \in V_{Word}\}}, \widetilde{w_i} = \frac{w_i}{avg\{w_{i'}|i' \in V_{DP}\}}. \quad (4)$$

Finally, X_{ij} is recalculated as

$$\widetilde{X_{ij}} = ceil(X_{ij} \cdot \widetilde{w_i} \cdot \widetilde{w_j}), \quad (5)$$

where $ceil(.)$ is a function that converts a floating number to the nearest integer.

Given the vector dimension, the vectors of words in V_{Word} and design patterns in V_{DP} are generated by GloVe[8] based on the entries co-occurrence counts matrix X. For training GloVe, we use the settings in [13], i.e., the number of iterations is 100, the initial learning rate is 0.05, and the model parameters $x_{max} = 100$ and $\alpha = 0.75$. Finally, the word and design pattern vectors are calculated as the sum of the "input" and "output" vectors generated by GloVe[9].

8. https://nlp.stanford.edu/projects/glove/

9. The source code and the learnt word and design pattern vectors can be accessed on https://github.com/WoodenHeadoo/dpword2vec.

4 Evaluation Settings

In this section, we present the experimental settings for evaluating the DPWord2Vec model, including evaluation protocols, baseline algorithms, evaluation metrics, and parameter settings of DPWord2Vec.

4.1 Evaluation Protocols

In this subsection, we demonstrate the strategy and dataset for evaluating DPWord2Vec.

Word similarity tasks are usually leveraged to evaluate the quality of word vectors in word embedding models [13], [18], [22], [23]. Generally speaking, two semantically relevant words should indicate that their vector representations are similar [22]. In DPWord2Vec, "word" means natural language word or design pattern. As we focus on the relationship between natural languages and design patterns, only the dp-word similarity is considered. This similarity can be estimated by calculating the cosine similarity of the word vector and the design pattern vector. To the best of our knowledge, there exist no publicly available datasets for dp-word similarity evaluation. Therefore, we build a new dataset of dp-word pairs with relatedness labels to address this issue[10].

Design Pattern Selection. At first, a list of design patterns is selected. To obtain a diverse list of design patterns, we select design patterns based on their frequencies, like the methods for word similarity datasets construction [18], [23]. The frequency of a design pattern means the number of documents in C that describe the design pattern. The 372 design patterns in V_{DP} can be grouped into five classes according to five frequency intervals: (0,10], (10,50], (50,400], (400,1500], and (1500,+∞). Except the first class which contains a relatively large number of infrequent design patterns, the other four classes have similar sizes, i.e., there are 34, 33, 33, and 34 design patterns in these classes respectively. We randomly sample ten design patterns from each class and get a list of 50 design patterns.

Pair Construction. Next, for each design pattern, we select a list of words to form pairs. Given a design pattern, if a word is randomly selected from V_{Word}, it is unlikely to be related to the design pattern. In other studies, word pairs are constructed by using WordNet synonym sets [18], [23]. However, there are no similar databases specified for design patterns as to our knowledge. Hence, we employ the frequency of co-occurrence to select words. The intuition is

10. We provide the dataset on https://github.com/WoodenHeadoo/dpword2vec.

if a design pattern and a word appear in the same document frequently, they are more likely to be relevant, then the word is more likely to be chosen. Concretely, given a design pattern, 40 non-duplicated words are randomly chosen based on a distribution, in which the probability of choosing a word is proportional to the number of documents containing both the word and the design pattern. Then we obtain 50×40 = 2,000 dp-word pairs and the number is comparable to those in [18] and [23].

Human Judgment. According to the last step of word similarity datasets construction [18], [23], [24], the relatedness between the design pattern and the word in each pair is manually labelled. To reduce the influence of personal biases, we recruit three graduate students to label the pairs. These participants all have bachelor's degrees majoring in computer science or software engineering and have been trained in object-oriented programming including design pattern relevant skills. They are also experienced with annotating software artifacts, such as evaluating the quality of the enriched API specifications and scoring the results of the code search algorithms.

Before labelling these pairs, all the participants go over the materials of the involving design patterns as a retrospect. When labelling, each dp-word pair is sent to each participant and he/she attempts to construct a context that the word is mentioned and associated with the design pattern. In this procedure, the participants are allowed to search for the texts that contain the design pattern and the word on the Internet to help them. If one still doubts whether such a context exists, the documents in C, in which the design pattern and the word co-occur, can serve as references. For each participant, a pair is labelled as "related" if the design pattern and the word can be associated in some certain contexts, and labelled as "unrelated" if they are hard to be linked or the meaning of the word is so general that the link seems to be too weak. The final label of a pair is "related" or "unrelated" if the participants can reach an agreement, i.e., they all label it as "related" or "unrelated". Otherwise, its final label is "somewhat related". That means, there exists some uncertainty but the relatedness is between "related" and "unrelated".

From the labelling process, we get some observations. Some pairs are consistently labelled as "related" since the word can describe the use scenario of the design pattern directly and the relationship between them can be easily imagined. For example, Publish/Subscribe is a messaging design pattern that provides instant notifications for distributed applications. The related words include "event" (the notifications are events), "channel" (notifications are broadcasted via the channel), and "endpoint" (the notification publishers and subscribers are all endpoints). Some pairs are related when considering the background of the entity that the word represents. For example, the word "wpf" refers to a programming framework. It is supposed to be related to the MVVM (Model View ViewModel) design pattern as it is a typical application of MVVM. For the pairs with the consistent label "unrelated", the association between the word and the design pattern is usually too weak to make sense. They may just be mentioned in a same document occasionally, for instance, Sharding - "excel", Iterator - "message", and Decorator - "plugin". The words whose meanings tend to be very general, such as "idea", "make", and "sometime", are also labelled as "unrelated" to any design patterns as it is difficult to specify a scenario that they can be related. Except for the consistently labelled ones, some pairs are controversial. For example, "dismiss" can represent a specific operation in the ViewController design pattern. However, it is also somewhat a general meaning word. Two participants judge it to be related to ViewController but the other one labels "unrelated". Hence, the final label is "somewhat related". To measure the degree of agreement among the participants, we calculate the Fleiss' Kappa. The value is 0.6421, which means a substantial agreement. Therefore, the labelling results are relatively reliable.

After the labelling process, 369 pairs (18.45%) are labelled as "related", 457 pairs (22.85%) are labelled as "somewhat related", and 1174 pairs (58.7%) are with the label "unrelated".

4.2 Baseline Algorithms

There exist several similarity methods to estimate semantic relatedness between natural language words. We take three categories of intensively used methods as baselines. This categorization can cover that adopted in [25].

4.2.1 Latent Semantics based Methods

In this category of methods, the words and design patterns are represented by latent variable vectors. Then the relatedness between a word and a design pattern can be measured by the cosine similarity[11]. This category includes Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA).

LSI (also known as Latent Semantic Analysis, LSA) is an unsupervised algorithm of analyzing the relationships between documents and terms by producing a set of latent semantic concepts [26]. It has been used in estimating semantic relatedness in source code [25]. In the evaluation, the input of LSI is the $term \times document$ matrix, in which an element represents the frequency of a term (word or design pattern) appearing in a document. Then the words and the design patterns are represented in a low-dimensional (latent) space by applying singular value decomposition. The dimension of the latent space is initially set as 10 and then gradually increased. During this process, the performance of LSI is evaluated. The value which achieves the best performance is retained and recorded. Finally, the dimension is set as 400[12].

LDA is a topic modeling technique that has been used for analysing software-specific data in several studies [20], [27], [28], [29]. To use LDA in the evaluation, each document in the corpus C is represented as a bag of words and design patterns without order. With the Gibbs sampling based implementation of LDA [30], each word or design pattern in a document is assigned to a topic. By considering the whole corpus, the words and the design patterns can

11. https://en.wikipedia.org/wiki/Cosine_similarity

12. In fact, the performance of LSI in terms of $NDCG@40$ and Spearman's ρ does not change much when the dimension is larger than 250. The details are shown on https://github.com/WoodenHeadoo/dpword2vec/blob/master/baselines/LSI.md.

be represented as probability distributions over topics. The topic number is set to 40 as it has been shown to be appropriate for the Stack Overflow dataset [28].

4.2.2 Co-occurrence based Methods

Co-occurrence based methods calculate the similarity (or distance) between a word and a design pattern directly based on their co-occurrences, including Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD).

PMI is an intuitive and computationally efficient relatedness method for massive corpora of textual data [31]. NGD is a semantic distance measure between words or phrases based on information distance and Kolmogorov complexity [32]. It has been verified to be effective in quantifying semantic relatedness between individual code terms (named Normalized Software Distance, NSD) [25]. Since NGD is a distance measure, the similarity can be obtained by negating the value of NGD. Both PMI and NGD take the frequency of a word (i.e., the number of documents containing the word), the frequency of a design pattern (i.e., the number of documents containing the design pattern), and the frequency of the co-occurrence (i.e., the number of documents containing both the word and the design pattern) in the corpus C as input, but calculate the measures in different ways.

4.2.3 Vector Space Model based Method

Another baseline is the Vector Space Model (VSM). Specifically, we use the TF-IDF (Term Frequency - Inverse Document Frequency) [33] schema to model the text. By multiplying each row of the $term \times document$ matrix (which is also the input of LSI) by the IDF value of the corresponding term, we obtain a matrix of TF-IDF values. Each row of the TF-IDF matrix can be regarded as the vector of the corresponding term (word or design pattern), which indicates the TF-IDF value of the term in each document. With these term vectors, the dp-word similarity can be also obtained by calculating the cosine similarity. Actually, the calculation of the IDF values is redundant in this case. Since the IDF weighting is operated on each entire term vector, the multiplied IDF values are eliminated automatically when calculating the cosine similarities. Therefore, this model is equivalent to represent a term with a row of the $term \times document$ matrix.

4.2.4 Software-Specific Method

In the evaluation, we consider a domain-specific method, WordSimSE, which aims to build WordNet like resources for software [24]. WordSimSE is a composite method that measures the similarity between terms by combining weighting strategy and co-occurrences. We use the WordSimSE method to calculate the dp-word similarities based on the corpus C. Moreover, there are three parameters to be clarified. According to the definition in [24], a word or a design pattern can be classified into one of the three groups: popular software tag, if it is a top 10% most frequent Stack Overflow tag; non-popular software tag, if it is a Stack Overflow tag but not in the top 10%; and ordinary term, otherwise. The three groups are weighted with three different parameters, namely 2.8, 2.0, and 1.4, which are also used in [24].

4.3 Evaluation Metrics

In our built dataset, each design pattern is paired with 40 words, which are labelled as "related", "somewhat related", or "unrelated" to the design pattern. We want to investigate whether the similarity scores given by the similarity methods could correspond with the labelled ones. To this end, we use two metrics for evaluation, namely NDCG and Spearman's rank correlation coefficient.

NDCG (Normalized Discounted Cumulative Gain) is a measure of ranking quality in information retrieval and employed in several software engineering tasks [34], [35], [36]. For each design pattern, a similarity method ranks the 40 words in descending order according to their similarity scores. The measure $NDCG@k$ is calculated as

$$NDCG@k = \frac{DCG@k}{IDCG@k}, DCG@k = \sum_{i=1}^{k} \frac{r_i}{\log_2(i+1)}, \quad (6)$$

where r_i denotes the degree of relevancy of the ith ranked word and its permissible values are 3 ("related"), 2 ("somewhat related"), and 1 ("unrelated"). $IDCG@k$ is the ideal value of $DCG@k$ that normalizes the measure into [0,1].

Spearman's rank correlation coefficient (Spearman's ρ) is a non-parametric measure of rank correlation which is usually used in the evaluations of word similarity tasks [13], [18], [23]. It represents the correlation between the ranks of the 40 words based on the similarity scores of a similarity method and the ranks based on the labelled relevance scores. However, there are only three unique labelled relevance scores in our dataset. Following [37], words with a same score are assigned with a same average fractional rank. Specifically, after ranking the 40 words according to the three labels, we assume that the first m_1 words are "related", the middle m_2 words are "somewhat related", and the last m_3 words are "unrelated". The rank of the "related" words is $\frac{1}{m_1} \cdot \frac{(1+m_1)m_1}{2} = \frac{m_1+1}{2}$, the rank of the "somewhat related" words is $m_1 + \frac{m_2+1}{2}$, and the rank of the "unrelated" words is $m_1 + m_2 + \frac{m_3+1}{2}$. Then the coefficient is calculated as

$$\rho = 1 - \frac{6\sum_{i=1}^{N} d_i^2}{N(N^2-1)}, \quad (7)$$

where $N = 40$, denotes the length of the rank list, and d_i is the difference between the two ranks of the ith word.

5 Evaluation Results

In this section, we investigate the following four research questions (RQs) to evaluate different aspects of DPWord2Vec.

5.1 RQ1: How do the settings of the parameters affect the performance of DPWord2Vec?

5.1.1 Motivation

The performance of DPWord2Vec may vary when using different settings. In this RQ, we investigate how DPWord2Vec performs under different values of the parameters, i.e., the dimension of the vectors, the size of context window for words, and the ratio of the weights of the two corpora.

5.1.2 Approach

Each of the three parameters is investigated independently. Specifically, we adjust the value of one parameter and analyse how the performance of DPWord2Vec changes. Meanwhile, the other two parameters keep fixed.

We change the value of the vector dimension (d) from 50 to 1,000, including 50, 100, 200, 300, 400, 500, 800, and 1,000. The value of the context window size for words (c) varies from 5 to 100, including 5, 10, 20, 30, 40, 50, 80, and 100. Moreover, we explore the importance of the description corpus and the crowdsourced corpus under different ratios of weights (r). The ratio $m : n$ indicates that each document in the description corpus and each document in the crowdsourced corpus are added into the final corpus for m and n times, respectively.

5.1.3 Results

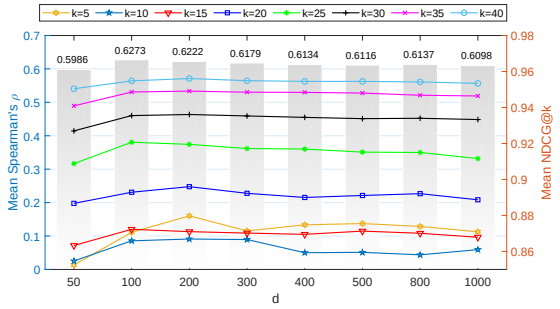
The results for the three parameters are presented respectively at follows.

Dimension of Vectors. The fold lines in Fig. 3a plot how the mean values of $NDCG@k$ change with different vector dimensions. For simplicity, we only show the results for k = 5, 10,..., 40. The bars in Fig. 3a show the mean values of Spearman's ρ on different vector dimensions. The settings of the other two parameters are c = 10 and r = 1:1. In Fig. 3a, we notice that all the fold lines have similar trends. The values of NDCG rise slightly when the vector dimension varies from 50 to 200 and then keep stable as the vector dimension increases further. Meanwhile, by referring to the bars, a similar trend can also be found on Spearman's ρ. In general, the performance of DPWord2Vec is not very sensitive to the vector dimension in terms of NDCG and Spearman's ρ.

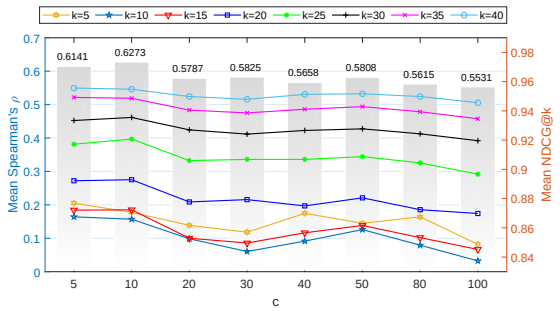
The dimension of the vector controls over the granularity of the representation of a word or a design pattern. A larger vector dimension tends to produce more fine-grained and detailed vector representations. However, the performance cannot further improve when the vector dimension is larger than 200. It may imply that the representations of words and design patterns reach the saturations at this vector dimension based on the current model and corpus.

Size of Context Window for Words. The values of NDCG and Spearman's ρ under different settings are presented in Fig. 3b as line chart and bar chart, respectively. The other two parameters are fixed at d = 100 and r = 1:1. As shown in the figures, both NDCG and Spearman's ρ all have an approximately descending trend as the context window size increases, especially from c = 10 to c = 20. The performance at c = 5 is comparable to that at c = 10. For example, $NDCG@40$ is 0.9556 at c = 5 and 0.9548 at c = 10, the former is slightly better; Spearman's ρ is 0.6141 at c = 5 and 0.6273 at c = 10, the later is slightly better. Generally, the descending trends are not very significant.

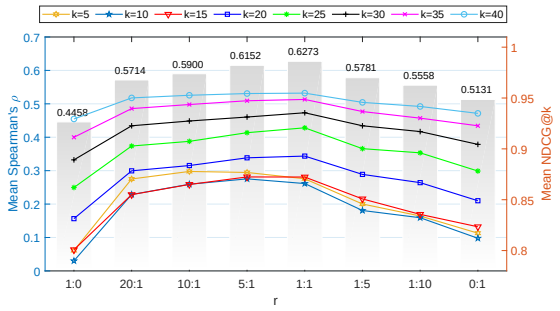
The context window size in DPWord2Vec only affects the context windows for words, it determines the number of surrounding words that a word is associated with. Too large context window size results in too many surrounding words that would diminish the syntactic information. It may lead to low-quality vector representations of words and design patterns, and then impairs the performance.



(a) Dimension of Vectors



(b) Size of Context Window for Words



(c) Ratio of Corpora Weights

Fig. 3: Mean NDCG and Spearman's ρ of DPWord2Vec under different parameter settings on the 50 design patterns.

Ratio of Corpora Weights. The results are shown in Fig. 3c. The other two parameters are set as d = 100 and c = 10. From the figures, we notice that the values of both NDCG and Spearman's ρ reach their peaks at r = 1:1, i.e., when the two corpora are directly mixed. The performance at r = 5:1 is the most similar one to that at r = 1:1. When changing the ratio, the performance drops and reaches the worst in the two directions at r = 1:0 and r = 0:1. That means, we will get bad results when using only one of the two corpora[13].

From the results, we can conclude that both the description corpus and the crowdsourced corpus are all indispensable for good performance. Although the description corpus is much smaller than the crowdsourced corpus, its effects cannot be neglected. The description corpus may stand for "quality" which supplies precise descriptions of design patterns, and the crowdsourced corpus stands for

13. Some words or design patterns may be out of the vocabulary when using only one corpus. In this case, the vectors are represented as random initial values. It may be a reason for the bad results. Nevertheless, it also implies that neither of the corpora is negligible.

"quantity" which provides rich textual data relevant to design patterns.

5.1.4 Conclusion

Generally, the performance of DPWord2Vec is not very sensitive to the dimension of vectors, but the settings of the context window size and the corpora weights affect the performance. To get a good performance, the context window size for words should not be too large, and the description corpus and the crowdsourced corpus should be balanced. The following experiments are all based on the settings that d = 100, c = 10, and r = 1:1.

5.2 RQ2: Does DPWord2Vec outperform the baseline algorithms in the dp-word similarity task?

5.2.1 Motivation

In this RQ, we explore whether DPWord2Vec can be superior to the baseline algorithms in dp-word similarity estimation.

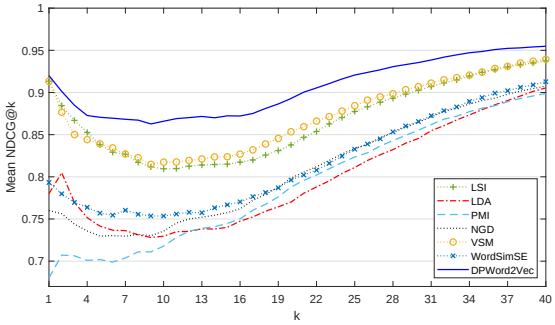
5.2.2 Approach

We compare DPWord2Vec against the six baseline algorithms, namely LSI, LDA, PMI, NGD, VSM, and WordSimSE, on our dp-word pair dataset. The two metrics, i.e., NDCG and Spearman's ρ, are applied for evaluation.

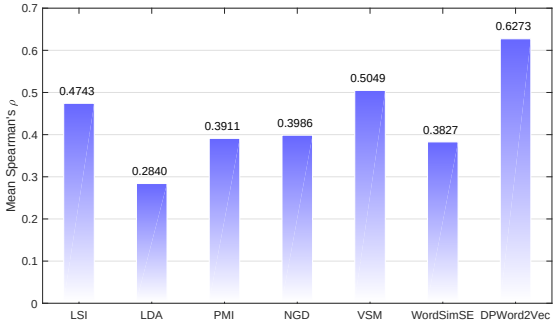
5.2.3 Results

Fig. 4a shows the mean values of $NDCG@k$ of the five algorithms and DPWord2Vec over the 50 design patterns on various k. Fig. 4b presents the averaged value of Spearman's ρ of these algorithms. As shown in Fig. 4a, DPWord2Vec almost outperforms all the baseline algorithms for all values of k. For example, $NDCG@40$ of DPWord2Vec is 0.9548, which outperforms those of LSI, LDA, PMI, NGD, VSM, and WordSimSE by 0.0173, 0.0494, 0.0559, 0.0472, 0.0155, and 0.0421, respectively. In Fig. 4b, DPWord2Vec outperforms LSI, LDA, PMI, NGD, VSM, and WordSimSE by 32.3%, 120.9%, 60.4%, 57.4%, 24.2%, and 63.9% respectively in terms of Spearman's ρ. As the metrics are only shown in mean values, we use Wilcoxon signed rank test [38] to investigate whether there are significant differences between the performance of DPWord2Vec and the baseline algorithms over the 50 design patterns. For $NDCG@40$, the p-values when comparing DPWord2Vec against the baseline algorithms are all less than 3e-6. For Spearman's ρ, the corresponding p-values are all less than 1e-7. That means, DPWord2Vec significantly outperforms the baseline algorithms in terms of NDCG and Spearman's rank correlation coefficient.

Among the baseline algorithms, LSI and VSM achieve better performance and the other four have somewhat comparable performance when considering NDCG and Spearman's ρ. We note that LSI and VSM are all based on the $term \times document$ matrix. It means that this way of text representation is relatively suitable for this task. The software specific method, WordSimSE, does not perform quite well in the evaluation. A possible reason is that there are differences between the software domain and the design pattern domain, as design patterns are universal solutions to recurring design problems and tend to be independent of specific software entities.



(a) NDCG



(b) Spearman's ρ

Fig. 4: Mean NDCG and Spearman's ρ of each baseline algorithm and DPWord2Vec on the 50 design patterns.

TABLE 2: The top 10 most related words to Record Set design pattern of each algorithm

LSI	LDA	PMI	NGD	VSM	WordSimSE	DPWord2Vec
recordset	querydef	recordset	recordset	recordset	recordset	recordset
record	recordset	tado	tado	record	row	row
querydef	clause	querydef	querydef	row	record	record
tado	statement	**jone**	row	**try**	value	clause
clause	row	pivot	pivot	tado	string	value
row	id	statement	record	**use**	execute	string
statement	**jone**	row	clone	value	**try**	array
exit	index	record	**jone**	**get**	id	pivot
try	record	clone	clause	**need**	**server**	index
modify	find	clause	statement	array	**get**	querydef

To gain more intuitions of how the algorithms perform, we give an example of ranked lists of these algorithms. Table 2 shows the top ten most related words to the design pattern Record Set [39] ranked by each algorithm. For DPWord2Vec, the ten words are all labelled as "related" or "somewhat related" to the design pattern Record Set. The top ten lists of the other algorithms all contain "unrelated" words, which are shown in boldface. For example, for LDA, PMI, and NGD, the top ten lists are contaminated by the noise word "jone". The word "jone" is a person name and usually used as an example of username when discussing database records in Stack Overflow (e.g., post #10050790). However, "jone" is not semantically related to Record Set. The top ten lists of LSI, VSM, and WordSimSE contain words with too general or vague meanings, e.g., "try", "get", and "use".

5.2.4 Conclusion

DPWord2Vec significantly outperforms the baseline algorithms on the dp-word similarity task in terms of NDCG and Spearman's ρ.

5.3 RQ3: Does the usage of the new context windows contribute to the performance of DPWord2Vec?

5.3.1 Motivation

In DPWord2Vec, we define new context windows for design patterns and words respectively (Section 3.3). In this RQ, we explore whether the usage of these context windows is an advisable choice to associate design patterns with words.

5.3.2 Approach

To investigate the effects of the new context windows, we replace them with the traditional fixed context windows used in Word2Vec [11] and repeat the experiments on the dp-word pair dataset. As the words and the design patterns are independent in the corpus C, we use two strategies to integrate words and design patterns into sequences, namely the *occurrence strategy* and the *shuffling strategy*, so that they can be handled by the traditional context windows.

The design pattern name occurrences strategy is to detect the occurrences of design pattern names in the text as design pattern tokens. This strategy is discussed in Section 3.3. The shuffling strategy is leveraged in a recent study to align words and APIs into a fixed context window [40]. Following [40], for a document doc, the words in $doc.Tokens$ and the design patterns in $doc.DPs$ are merged and randomly shuffled for ten times to produce ten token sequences (containing both words and design patterns).

Moreover, we also consider two other strategies which represent design patterns in higher levels rather than token level. They are from Doc2Vec [41] and Category enhanced Word Embedding (CeWE) [42], respectively. The original Doc2Vec aims to embed words and paragraphs or documents into vector spaces. Based on this model, we regard a design pattern as a document-level term to learn its vector representation. Specifically, the vector of each document in Doc2Vec is substituted with the vector of the design pattern which is contained in the document. Each design pattern in V_{DP} always keeps a unique vector even if it appears in different documents. However, a document may contain multiple design patterns. In this case, its word tokens ($doc.Tokens$) are duplicated multiple times so that each duplicate can be combined with a design pattern. Recently, Nguyen et al. have used the same approach to produce the vector representations of APIs and words [43].

Likewise, CeWE can learn the vector representations of words as well as categories. A category indicates a label or a classification of documents. A document may belong to multiple categories. In this study, we regard each design pattern as a category. In this way, design patterns are also associated with words in document level and their vectors can be obtained accordingly.

For all the strategies above, the parameters, including the dimension of the vectors, the size of context window, the initial learning rate, and the number of iterations, are the same as in Section 3.4. As introduced in [42], the parameter λ of CeWE is set to be $1/(2 \cdot c + 1)$, where c is the size of context window.

5.3.3 Results

The results are shown in Fig. 5a and Fig. 5b in terms of NDCG and Spearman's ρ, respectively. As shown in the figures, we notice that the performance of DPWord2Vec with the *occurrence strategy* (Occ.) is poor. For example, the Spearman's ρ is 0.1315, even worse than all the baseline algorithms in Section 5.2. DPWord2Vec with the *shuffling strategy* (Shuff.), and the strategies of Doc2Vec and CeWE, achieve comparable performance. Among them, the *shuffling strategy* tends to be slightly better than the other two, but still surpassed by DPWord2Vec with the new context windows. According to Wilcoxon signed-rank test, the differences between the performance of the default DPWord2Vec and that with the other strategies on $NDCG@40$ and Spearman's ρ are statistically significant (p-values are all less than 1e-5).

The drawback of the *occurrence strategy* is obvious. As the design pattern names tend to be sparse in the text, it is hard to mine the relationships between words and design patterns adequately by leveraging the context windows. With regard to the *shuffling strategy*, it may break the structure of the natural language sentences and do harm to the capture of semantic relationships. Moreover, the shuffling process will significantly increase the size of the corpus (almost ten times the original one) which results in extra computation complexity.

The two document-level strategies, i.e., that from Doc2Vec and CeWE, have similar mechanisms. The core is that, in each document, the vectors of the design patterns are integrated with the vectors of the surrounding words in a context window to predict the central one. Hence, design patterns can be deemed to be contained in the context of words in some way. However, there exists no similar context for design patterns and the design patterns in a document are not predicted by the vectors of the involving words. Compare to these strategies, the new context windows can build stronger ties between design patterns and words.

5.3.4 Conclusion

DPWord2Vec with the new context windows can achieve better results than the variants with the two serializing strategies and the two document-level strategies. Thus, the usage of the new context windows does contribute to the performance of DPWord2Vec.

5.4 RQ4: Does the weighting strategy contribute to the performance of DPWord2Vec?

5.4.1 Motivation

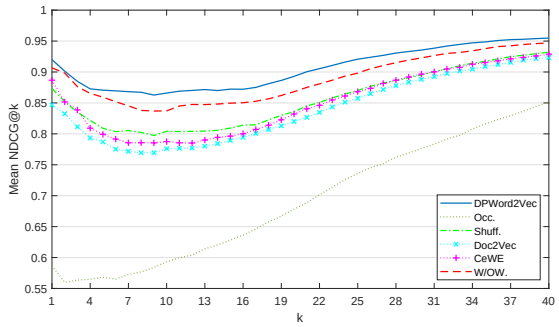
A weighting strategy is applied in the training phase of DPWord2Vec (Section 3.4). To verify whether this strategy is redundant, we set up this RQ.

5.4.2 Approach

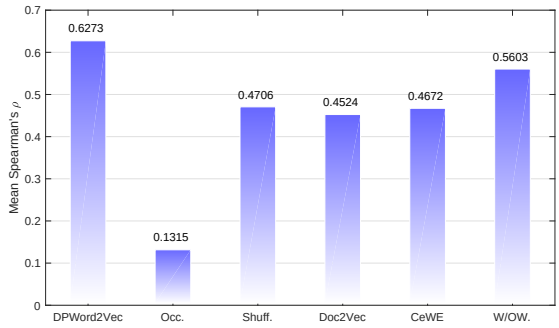
We construct a variant of DPWord2Vec by removing the weighting strategy. Then the performance of the variant is compared against that of the default DPWord2Vec.

5.4.3 Results

The results are also presented in Fig. 5a and Fig. 5b (W/OW.). In Fig. 5a, we observe that there are minor effects on $NDCG@k$ with small k after removing the weighting strategy. The differences are obvious when k is larger than five. When considering all the 40 words for each design pattern, the values of $NDCG@40$ and Spearman's ρ after



(a) NDCG



(b) Spearman's ρ

Fig. 5: Mean NDCG and Spearman's ρ of the variants of DPWord2Vec on the 50 design patterns. Occ. = DPWord2Vec with the occurrence strategy, Shuff. = DPWord2Vec with the shuffling strategy, Doc2Vec = DPWord2Vec with the strategy of Doc2Vec, CeWE = DPWord2Vec with the strategy of CeWE, W/OW. = DPWord2Vec without the weighting strategy.

removing are respectively 0.9471 and 0.5603, which are all worse than the original ones, i.e., 0.9548 and 0.6273. As the mean values seem to be close, we perform Wilcoxon signed-rank test on $NDCG@40$ and Spearman's ρ. The p-values are respectively 2.72e-3 and 1.05e-5, which indicates the differences are significant according to the $p < 0.05$ standard. Moreover, we quantify the magnitude of the difference of performance by analysing the effect size. Specifically, Cohen's d [44] is calculated to measure the differences between the means of the metrics with and without the weighting strategy. The results for $NDCG@40$ and Spearman's ρ are 0.2959 and 0.6087, which indicate a small-medium effect size and a medium-large effect size [45], respectively. That means, the effect of the weighting strategy on the performance is not negligible.

Based on the results, we note that DPWord2Vec achieves better performance with the weighting strategy, especially in terms of $NDCG@k$ with $k > 5$. Without the weighting strategy, the irrelevant but frequent words, such as "get" and "case", may be included in the top k list with a relatively large k and ranked ahead of the ones which are labelled as "related". The weighting strategy could effectively weaken the relationships between the design patterns and these words, thus improves the performance.

5.4.4 Conclusion

DPWord2Vec can benefit from the weighting strategy for measuring dp-word similarity.

6 Application I: Design Pattern Tag Recommendation

Many software information sites allow developers to label their posts with tags, such as Stack Overflow, Ask Ubuntu, and Freecode. Tags are short descriptions within a few words long that are provided as metadata to classify, identify, and search software objects in these sites [46]. To improve the quality of tags in software information sites, a series of automatic tag recommendation methods have been proposed to recommend appropriate tags for new posts based on existing tag candidates [47], [48], [49], [50], [51]. In this application, we consider a design pattern specific tag recommendation task that recommends design pattern tags for design pattern relevant posts. That is, each recommended tag is a design pattern. By the recommendations, the synonymous design pattern tags could be better avoided, which results in better information organization and retrieval for design pattern relevant posts.

6.1 Common Methods for Tag Recommendation

Actually, the design pattern tag recommendation task can also be accomplished by general tag recommendation methods. We briefly introduce the methods for tag recommendation.

The main intuition of the existing tag recommendation methods is to use the historical information of tag assignments to recommend tags for new posts. Concretely, the tag recommendation methods analyse the existing posts and their tags in a software information site, and then infer the relationship between a tag and a word or a whole post. When a new post is coming, the same analysis process is deployed on this post with the inferred results and each tag is given a likelihood score. The top few tags with the highest likelihood scores will be recommended. By restricting the tags to design pattern tags, i.e., each tag represents a design pattern, these methods are directly applied in the design pattern tag recommendation task.

6.2 Design Pattern Tag Recommendation based on DPWord2Vec

In this part, we explain how to recommend design pattern tags by leveraging DPWord2Vec.

With DPWord2Vec, design patterns and natural languages are associated. We can use these associations for design pattern tag recommendation. As the content of a post is a typical document that contains multiple words, to recommend design pattern tags for a post, the relationship between a design pattern and a document should be built based on the word and design pattern vectors. Therefore, we adopt the text semantic similarity [52] to measure the relatedness between a design pattern and a set of words:

$$Sim(Words, dp) = \frac{1}{2}[\frac{\sum_{w \in Words} IDF(w) \cdot Sim(w, dp)}{\sum_{w \in Words} IDF(w)} + \max_{w \in Words} Sim(w, dp)], \quad (8)$$

where $IDF(w)$ is the inverse document frequency[14] value of the word w in the corpus C and $Sim(w, dp)$ is the vector cosine similarity between w and the design pattern dp.

Generally, given a new design pattern relevant post, there are three steps for design pattern tag recommendation:

1) Preprocess and tokenize the textual description of the post following the procedures in Section 3.2.
2) For each design pattern tag in the tag candidate set, calculate the similarity between the design pattern and the post as Formula 8.
3) Rank the design pattern tags in descending order according to their similarities and recommend the top k design pattern tags.

6.3 Evaluation on Design Pattern Tag Recommendation

6.3.1 Motivation

In the evaluation, we try to explore whether the DPWord2Vec-based method performs better than the common tag recommendation methods on the design pattern tag recommendation task.

6.3.2 Approach

To evaluate the effectiveness of the DPWord2Vec-based method, we compare it against the state-of-the-art tag recommendation algorithms on a real-world dataset. We detail the strategies for evaluation, the constructed datasets, the state-of-the-art tag recommendation algorithms, and the leveraged metrics as follows.

Strategies. As to our knowledge, there are two software information sites in which design patterns are broadly discussed: Stack Overflow and Software Engineering[15]. However, on one hand, the posts in Stack Overflow have been leveraged by DPWord2Vec, it is inappropriate to use them to evaluate DPWord2Vec again. On the other hand, the amount of design pattern relevant posts in Software Engineering is relatively small (less than 3,000, a dataset of tag recommendation usually contains more than 13,000 posts [47], [48], [49], [50], [51]), it may be detrimental for the other tag recommendation algorithms to train proper models based on these posts. Therefore, the main strategy for evaluation is to use the Software Engineering posts for testing, and use the Stack Overflow posts to train tag recommendation models.

Datasets. We download the Stack Overflow posts (from Aug. 2008 to Dec. 2017) and the Software Engineering posts (from Sep. 2010 to Mar. 2019) to construct the datasets. Before that, the design pattern tags should be detected. At first, we construct the regular expressions for the names of each design pattern in V_{DP}. Specifically, each design pattern name is split into word(s), i.e., $word_1$, $word_2$, ... , $word_n$, and the regular expression is written as "$word_1$-?$word_2$...-?$word_n$(-pattern)?" (as words can only be separated by hyphens in tags). In this way, the tags like "active-record", "activerecord", and "active-record-pattern" can all be matched with the design pattern name "active record". Next, all the tags of these posts are extracted and a tag is mapped to a design pattern if it matches with a name of the design pattern via the corresponding regular expression. Then, we manually review each mapped tag if it has a description in the corresponding software information site to filter out false-positive tags that do not denote the design patterns. At last, multiple tags are merged into one tag if they are mapped to the same design pattern. Finally, 94 and 36 design pattern tags are detected in Stack Overflow and Software Engineering, respectively. In this way, the design pattern tags of the two sites are unified and these tags have a one to one correspondence with the design patterns.

With the design pattern tags, we construct two datasets: a dataset for training the common tag recommendation models and a dataset for testing the common models and the DPWord2Vec-based model. To build the training set, we extract the Stack Overflow posts that contain the design pattern tags but discard the tags appearing in less than 50 posts as they are less interesting and less useful to serve as representative tags [47]. For the test set, we extract the Software Engineering posts that contain the design pattern tags but discard the tags not appearing in the training set as they cannot be recommended by the common tag recommendation algorithms. Finally, the training set contains 176,427 Stack Overflow posts and 74 design pattern tags which are used as candidates, the test set contains 2,986 Software Engineering posts and 35 design pattern tags[16]. Like the training set here, the crowdsourced corpus, which is for training the design pattern and word vectors, is also constructed based on the Stack Overflow posts. It should be noted that they are distinct. The crowdsourced corpus consists of the posts with at least one design pattern name appearing in the titles, bodies, or tags. It involves 210 design patterns in total. In contrast, the training set only cares about the posts containing design pattern tag(s). The latter can be roughly covered by the former.

According to the settings above, the common tag recommendation models are trained on the Stack Overflow posts containing the design pattern tags. Meanwhile, our DPWord2Vec-based model relies on the design pattern and word vectors learnt form the corpus C. In other words, these models do not have a consistent training set. To achieve unbiased comparisons, we conduct another part of evaluation in which the corpus C is also used for training the common tag recommendation models. Specifically, each document in C is regarded as a post and each design pattern in a document is regarded as a design pattern tag. Then, all the 372 design patterns in V_{DP} serve as candidates.

State of the Arts. To the best of our knowledge, there are three common tag recommendation algorithms, TagMulRec [49], EnTagRec++ [50], and FastTagRec [51], shown to be the state-of-the-art on software information sites. Similar to word embedding models, FastTagRec represents words as vectors and recommends tags using neural network-based classification. Given a new post, TagMulRec first locates the posts that are semantically similar to it, and then exploits multi-classification to produce a ranked tag list. EnTagRec++ integrates the historical tag assignments and

14. https://en.wikipedia.org/wiki/Tf-idf
15. https://softwareengineering.stackexchange.com/
16. The training and test sets, as well as the original tag - design pattern mappings are available on https://github.com/WoodenHeadoo/dpword2vec.

TABLE 3: The results on the design pattern tag recommendation task (Stack Overflow posts for training TagMulRec and FastTagRec, the 74 design pattern tags in Stack Overflow as candidates)

	Baseline	TagMulRec	FastTagRec	DPWord2Vec
$Recall@5$	0.7369	0.5279	0.8167	**0.8399**
$Precision@5$	0.1618	0.1123	0.1786	**0.1837**
$F1-score@5$	0.2625	0.1838	0.2901	**0.2984**
$Recall@10$	0.7369	0.6954	0.8658	**0.9230**
$Precision@10$	0.0809	0.0749	0.0952	**0.1017**
$F1-score@10$	0.1448	0.1345	0.1704	**0.1820**

TABLE 4: The results on the design pattern tag recommendation task (corpus C for training TagMulRec and FastTagRec, all the 372 design patterns as candidates)

	Baseline	TagMulRec	FastTagRec	DPWord2Vec
$Recall@5$	0.7358	0.5559	0.8322	**0.8399**
$Precision@5$	0.1615	0.1183	0.1826	**0.1837**
$F1-score@5$	0.2620	0.1936	0.2963	**0.2984**
$Recall@10$	0.7369	0.7040	0.8895	**0.9224**
$Precision@10$	0.0809	0.0758	0.0978	**0.1017**
$F1-score@10$	0.1448	0.1361	0.1750	**0.1819**

the information of users for tag recommendation. However, EnTagRec++ cannot be applied here as the training set and the test set are from different sites which do not share the same group of users. Therefore, we only take TagMulRec and FastTagRec for comparisons.

In addition, with the concern that the design pattern names may appear in the posts explicitly, we deploy a baseline method which leverages the occurrences of design patterns. Specifically, the design pattern names of each design pattern in the tag candidate set are searched in the Software Engineering posts (the test set) by using the regular expressions (as discussed in Section 3.1). A post is supposed to contain a design pattern tag if one of the design pattern names appears in the title or body of the post. Since the common tag recommendation methods only provide likelihood scores for ranking the candidate tags, for this baseline method, the design pattern tags are also sorted according to the numbers of design pattern occurrences for comparability. If there are no or not enough design pattern occurrences found in the post, the design pattern tags are sorted in alphabetical order.

Metrics. The recommending strategy of all the algorithms above is to provide a rank list of candidate design pattern tags and recommend the top k ones. To evaluate the recommendations, we exploit three metrics, $Recall@k$, $Precision@k$, and $F1-score@k$, which are usually used to evaluate tag recommendation systems on software information sites [49], [51]. In particular, the sample-wise metrics are calculated as

$$Recall@k_i = \frac{|RankList_i^k \cap Tag_i|}{|Tag_i|} \tag{9}$$

and

$$Precision@k_i = \frac{|RankList_i^k \cap Tag_i|}{k}, \tag{10}$$

where Tag_i and $RankList_i^k$ are the set of real design pattern tags and the set of top k recommended design pattern tags for the ith posts in the test set, respectively. By combining $Recall@k_i$ and $Precision@k_i$,

$$F1-score@k_i = \frac{2 \cdot Recall@k_i \cdot Precision@k_i}{Recall@k_i + Precision@k_i}. \tag{11}$$

Then the set-wise metrics $Recall@k$, $Precision@k$, and $F1-score@k$ are respectively the average values of the sample-wise metrics in Formulas 9, 10, and 11 over all the posts in the test set. According to the literature [47], [48], [49], [50], [51], k is set to 5 and 10.

6.3.3 Results

As introduced before, the evaluation contains two parts. In the first part, the Stack Overflow posts with the design pattern tags are used for training the TagMulRec model and the FastTagRec model, the Software Engineering posts are used for testing all the models. The tag candidate set for recommendation includes the 74 design pattern tags appearing in these Stack Overflow posts. The results are shown in Table 3. The best result on each metric is shown in boldface. As shown in the table, the DPWord2Vec-based method achieves much better performance than TagMulRec, i.e., over 30% improvements on all metrics. When comparing against FastTagRec, the improvements are not so apparent, i.e., all within 10%. We perform Wilcoxon signed-rank test on sample-wise metrics of all the 2,986 posts and the p-values on the six metrics are all less than 0.0025 when comparing DPWord2Vec against FastTagRec. That means, the DPWord2Vec-based method significantly outperforms FastTagRec in statistics.

In the second part, we train the TagMulRec model and the FastTagRec model using the corpus C and test all the models with the Software Engineering posts. The candidates are changed to all the 372 design patterns in V_{DP}. Table 4 presents the evaluation results. From the table, we notice that the performance of TagMulRec and FastTagRec improves on all the metrics contrast to the previous ones, but is still not as good as that of the DPWord2Vec-based method. The DPWord2Vec-based method is relatively stable as the results are almost unchange when involving more design pattern tag candidates. According to the results of Wilcoxon signed-rank test, the differences on $Recall@5$, $Precision@5$, and $F1-score@5$ are not significant, i.e., the p-values are 0.22, 0.44, and 0.19, respectively. However, the DPWord2Vec-based method still significantly outperforms FastTagRec when recommending ten design pattern tags, i.e., p-values on $Recall@10$, $Precision@10$, and $F1-score@10$ are all less than 1e-6. It implies that the DPWord2Vec-based method benefits from not only a comprehensive corpus but also an appropriate algorithmic model.

As shown in Tables 3 and 4, it is surprising that the performance of the baseline method is better than that of TagMulRec on all metrics, although surpassed by that of FastTagRec and the DPWord2Vec-based method. That means, to detect the design pattern occurrences is also effective for design pattern tag recommendation to some degree. From the perspective of Recall, the names of a part of the design patterns serving as tags appear in the text of the posts as well. But it does not achieve a quite ideal coverage. From the perspective of Precision, an occurrence of a design pattern name in a post does not necessarily mean

that it is also a tag of the post, as the design pattern may be not the main focus or the mentioned design pattern name is ambiguous. Comparing Table 4 against Table 3, we notice that the baseline method has minor changes in performance when enlarging the tag candidate set. The reason is that the newly involved design patterns appear rarely in the posts.

Generally, the performance of the DPWord2Vec-based method is relatively close to that of FastTagRec. Nevertheless, there are some advantages of our method. On the one hand, the DPWord2Vec-based method is more efficient than FastTagRec. As DPWord2Vec is based on the GloVe model, the time complexity for calculating and updating the gradients is usually $O(d(|C|^{1/\alpha} + |DPTags|^{1/\beta}))$ for some α, $\beta > 1$ [13], where d denotes the dimension of the vectors, $|C|$ denotes the total number of word tokens, and $|DPTags|$ denotes the total number of design pattern tag occurrences in the training set. As $|DPTags|$ ought to be much smaller than $|C|$, the time complexity can be written as $O(d \cdot |C|^{1/\alpha})$. For FastTagRec, the time complexity is $O(d \cdot |C| \cdot log(|DPCands|))$ [51], where $|DPCands|$ denotes the size of the design pattern tag candidate set. Hence, the DPWord2Vec-based method is more scalable when involving more posts for training ($|C|$ gets larger). Moreover, enlarging the tag candidate set will make the model of FastTagRec more complex, but not explicitly increase the model complexity of the DPWord2Vec-based method. On the other hand, the DPWord2Vec-based method is more understandable. FastTagRec is essentially a classification model. It regards each design pattern tag candidate as a class and recommends tags by training the classifier. However, the classifier is somewhat a black-box for the users. In contrast, DPWord2Vec represents the elements of the natural language and the tag candidates as vectors, and ranks the tags according to the similarities between them and the post. It tends to be more intuitive and acceptant for humans. Moreover, by exploring the sentences or phrases with high similarities to the tags, the users could understand the motivation of the recommendation better.

6.3.4 Conclusion

In the design pattern tag recommendation task, the DPWord2Vec-based method performs better than TagMulRec and FastTagRec in terms of Recall, Precision, and F1-score, even when they are provided with the same data for training. This shows that the learned word and design pattern vectors could better express the relationships between a post and a design pattern.

7 Application II: Design Pattern Selection

When developing a software (sub)system, the developer(s) may be willing to leverage design patterns to facilitate the development process. This is called a design problem. However, there exist a large number of design patterns [7] and determining the applicability of these design patterns heavily depends on the experience of a developer [53]. It is usually difficult to find the right design pattern(s) for a given design problem especially for novice developers [8]. To resolve this problem, several studies focus on selecting appropriate design pattern(s) automatically based on the textual description of the design problem [8], [54]. The textual description is a short text that may depict the main features, requirements of the (sub)system, or how it works.

In this application, we attempt to solve this design pattern selection problem by leveraging the learnt word and design pattern vectors. Comparing to the previous task, i.e., design pattern tag recommendation, design pattern selection is usually a more challengeable task. In the previous task, a post may involve explicit characteristics of design patterns, e.g., design pattern names. However, in this task, the description of the design problem cannot contain such information as the suitable design pattern(s) is assumed to be unknown. The semantic meaning of the description should be explored and it should match the application scenarios of the selected design pattern(s).

7.1 General Method of Design Pattern Selection

In this part, we introduce the general framework of design pattern selection in the existing studies.

The existing design pattern selection approaches usually use the problem definition of a design pattern as the oracle for design pattern selection [8], [54]. The problem definition describes what problems the design pattern solves and where the design pattern can be applied. For example, in the GoF book, the problem definition contains the intent, motivation, and applicability sections [8]. Given a design problem description and a collection of design patterns, the design pattern selection procedure can be detailed in the following three phases [8], [54].

Vectorizing the Documents. The documents, i.e., the design problem description and the problem definitions of design patterns, are preprocessed and vectorized by leveraging the vector space model, in which each document is presented as a feature vector and each feature indicates the weight of a word in the document.

Determining the Design Pattern Class. This phase aims to preliminarily find a set of design patterns that are likely to be right for the design problem. It is motivated by the expert classification of design patterns. For example, the 23 design patterns in GoF are divided into three classes, i.e., Creational Patterns, Structural Patterns, and Behavioral Patterns [2], and each class focuses on one type of design problems. Therefore, the goal is to determine the most suitable design pattern class for the design problem. With this phase, the design pattern selection process can leverage the expert classification information besides the similarity between the design problem and the oracle of a design pattern. Hence, this phase is a reinforcement for the similarity-based selection and the accuracy is expected to be improved.

To determine the design pattern class, text categorization methods are applied to these vectorized text documents. For example, [8] leverages supervised learning methods to build a classifier for textual descriptions based on the expert classes of design patterns. Then the design problem description is classified into a class by the classifier and the design patterns in this class are delivered to the next phase. Similarly, [54] uses clustering methods to group the problem definitions of design patterns and the design problem description into multiple clusters. This partition may be not consistent with the expert classification, but the numbers of classes (or clusters) in the two partitions are equal. The

design patterns whose problem definitions are in the same cluster with the design problem description are retained for further selection.

Suggesting the Design Pattern(s). With the determined class of design patterns, the appropriate design pattern(s) is further suggested based on the similarities between the design problem description and the problem definitions of design patterns. Concretely, the ith design pattern in the determined class is suggested if

$$\begin{cases} |S_i| > \theta_1 \\ |S_i - S_{max}| \leq \theta_2 \end{cases}, \tag{12}$$

where S_i is the similarity between the problem definition of the ith design pattern and the design problem description, S_{max} is the maximum among the similarity S_j corresponding to each design pattern in the determined class, and θ_1 and θ_2 are thresholds that should be specified manually. We note that more than one design patterns may be selected finally. The result relies on the values of the thresholds.

7.2 Refined Design Pattern Selection Method based on DPWord2Vec

With the learnt design pattern and word vectors, we show how to refine the existing design pattern selection method.

As to the depictions above, the design pattern selection method depends on the expert classification of design patterns. However, this classification may involve inconsistencies and anomalies [8]. In other words, the classification may not be fully reflected by the problem definitions of the design patterns. As a result, the determined class may be unreliable. Therefore, we modify the second phase, i.e., Determining the Design Pattern Class, by leveraging the learned word and design pattern vectors to refine the design pattern selection method.

There are three steps for the modified phase:

1) Preprocess and tokenize the design problem description following the procedures in Section 3.2.
2) For each design pattern candidate, calculate the similarity between the design pattern and the design problem description as Formula 8.
3) Perform k-means clustering [55] on the design pattern candidates to group them into the "relevant" class and "irrelevant" class based on their similarities with the design problem description. The initial centroids of the two clusters are the maximum and minimum of the similarities, respectively. The "relevant" class is considered as the candidate design pattern class for the design problem.

This new phase doesn't use any information of the expert classification but leverages the relatedness between the design problem and design patterns inferred from the word and design pattern vectors. The design patterns with very weak relatedness to the design problem are unlikely to be the appropriate ones and eliminated, the rests are retained for further selection. Except for the second phase, the first and third phases of the method keep unchanged.

7.3 Evaluation of the DPWord2Vec-based Method

7.3.1 Motivation

To investigate whether the refined method based on DPWord2Vec is effective, we set up this evaluation.

7.3.2 Approach

We compare the refined method based on DPWord2Vec against the existing ones on design pattern selection benchmarks. In the following parts, we depict the benchmarks, the methods for comparison, the evaluation metrics, and the settings of all the methods, respectively.

Benchmarks. The benchmarks we use are the same as those used in [54], which involve 80 design problems and three design pattern collections, namely GoF [2], Security [56], and Douglass [57]. The GoF collection includes 23 object-oriented design patterns which are divided into three classes. The Security collection includes 46 design patterns used in integrating security systems and presented in eight classes. There are 34 real-time system relevant design patterns in the Douglass collection and they have been divided into five classes. The numbers of design problems corresponding to the three collections are 30, 30, and 20, respectively. For each design problem, only one design pattern in the collection is regarded as correct[17].

Following [54] and [8], for each collection, the evaluation is deployed independently. Only the design patterns in this collection are considered as the original candidates for selection.

State of the Arts. As to our knowledge, there are two studies, [54] and [8], that propose completely automatic design pattern selection methods based on publicly available textual descriptions of design patterns. The methods in these two studies all follow the three-phases framework mentioned above. In this evaluation, we take them for comparison.

Metrics. Following [54] and [8], the design pattern selection methods are evaluated by the RCDDP (Ratio of Correct Detection of Design Pattern) metric, which is calculated as

$$RCDDP = \frac{1}{N}\sum_{i=1}^{N}\frac{|SDP_i \cap CDP_i|}{|SDP_i|}, \tag{13}$$

where N is the number of design problems for the design pattern collection, CDP_i is the set of correct design pattern(s) to solve the ith design problem (contains only one design pattern in the dataset), and SDP_i is the set of suggested design pattern(s) by the design pattern selection method.

As to the definition above, the RCDDP metric depends on the values of the thresholds θ_1 and θ_2, as they will determine which design pattern(s) is finally suggested, i.e., SDP_i. It may make the comparisons complicated, since the appropriate values of the thresholds for different design pattern selection methods may be not unified. Actually, our refined method only modifies the phase of determining the design pattern class, but does not deal with the settings of the thresholds. Without losing the reasonability, we leverage

17. The 80 design problems and the corresponding correct design patterns can be found on https://github.com/WoodenHeadoo/dpword2vec.

another metric for evaluation, namely MRR (Mean Reciprocal Rank) [58], which is not affected by the thresholds. MRR is a standard evaluation metric in information retrieval and used in several software engineering related studies [59]. Specifically,

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_c^i}, \tag{14}$$

where $rank_c^i$ denotes the position of the correct design pattern to the ith design problem in the rank based on the similarities in the third phase. The expression $1/rank_c^i$ is called as reciprocal rank. If the correct design pattern is eliminated in the second phase, then the reciprocal rank is 0. As to the definition, the value of MRR is low if most of the correct design patterns are omitted; and high if the irrelevant design patterns ranked before the correct ones are eliminated. Therefore, MRR is able to evaluate the candidate design pattern class produced in the second phase.

Settings of the Methods. The methods in [54] and [8] are more like frameworks rather than concrete algorithms. That means, the concrete algorithms for each step should be specified according to the realities. Therefore, we unify the settings for all methods and leverage the moderate ones that perform best in the most cases according to the results in [54] and [8].

Specifically, the TF-IDF technique is used for the vectorization of the documents in the first phase. In the second phase, the improved global feature selection scheme [60] is used to reduce the dimension of the document vectors. The support vector machine [55] classification algorithm and fuzzy c-means clustering [61] algorithm are leveraged to determine the candidate design pattern class for the method in [8] and the method in [54], respectively. The number of classes (clusters) is consistent with that of the expert classification in each design pattern collection. In the third phase, the cosine similarity is applied to measure the correlation between the vectorized problem definitions of design patterns and design problem descriptions.

For the refined method, the TF-IDF technique and cosine similarity are also used in the first and third phases, respectively. But the second phase is replaced by the modified one.

According to [54] and [8], the effective values of the thresholds and the number of features (dimension of the document vectors after feature selection) rely on the design pattern collections. Hence, we attempt to find the most suitable settings for each collection and report the optimal results. For the methods in [54] and [8], we try various feature numbers from 50 to the vocabulary size at an interval of 50 and the best one in terms of MRR is recorded. Then, for each method, we find the highest value of RCDDP by traversing all the combinations of θ_1 and θ_2 from the range $\{0, 0.1, 0.2,..., 1.0\}$ and the range $\{0, 0.01, 0.02,..., 0.10\}$ [8], respectively.

7.3.3 Results

The metric values and the corresponding parameter settings are displayed in Table 5. As shown in the table, the refined method achieves the best performance on all three collections. Averaging across the three collections, the refined method outperforms the method in [54] (M1) by 6.3% and 6.5% in terms of RCDDP and MRR, respectively. The performance of the method in [8] (M2) is overall unsatisfactory. For example, the refined method improves M2 by over 70% in terms of the mean value of MRR.

TABLE 5: The results and parameter settings on the design pattern selection task

GoF					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.5333	0.6368	(0, 0)	950	16.67%
M2	0.3333	0.3417	(0, 0)	950	63.33%
Refined	**0.5667**	**0.6806**	(0, 0)	-	**13.33%**
Security					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.8000	0.8278	(0, 0)	900	13.33%
M2	0.4333	0.4500	(0, 0)	200	53.33%
Refined	**0.8667**	**0.9111**	(0, 0)	-	**3.33%**
Douglass					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.6000	0.6917	(0, 0)	600	15.00%
M2	0.4583	0.5542	(0, 0.08)	650	30.00%
Refined	**0.6225**	**0.7058**	(0, 0.1)	-	**5.00%**

M1 = the method in [54], M2 = the method in [8], Refined = the refined method based on DPWord2Vec

The possible reason for the bad results of M2 is that too many correct design patterns are eliminated when determining the design pattern class. To show this observation, for each method on each collection, we calculate the ratio of cases in which the correct design pattern is eliminated after the second phase. The results are also shown in Table 5, namely Erroneously Eliminating Ratio (EER). From the results, we notice that the EERs of M2 are very high for all the collections. For example, the EER of M2 is 63.33% for the GoF collection. That means, the correct design patterns of the 19 among the 30 design problems are mistakenly eliminated. Meanwhile, the EERs of the refined method are the lowest among all the methods on the three collections.

Notably, the performance of the refined method is not much better than that of M1 on the Douglass collection. The MRR values are respectively 0.7058 and 0.6917, which are quite similar. Generally, the quality of the learnt design pattern vectors relies on the design pattern relevant documents. However, in the corpus C, the number of documents relevant to each design pattern in the Douglass collection seems to be too few. Counting all the 20 design patterns mentioned in the benchmarks, nine of them relate to less than 10 documents each, eight design patterns occupy 10 to 49 documents each, and each of the other three ones involves 50 to 73 documents. It could be the reason for the nonsignificant improvement in the Douglass collection.

The main difference among M1, M2, and the refined method is the way of determining the candidate design pattern class. M2 chooses one class of the expert classification as the candidate class but this way does not work well. M1 does not completely follow the expert classification but use it during the feature selection. The performance of M1 is much better than that of M2, but not as good as that of the refined method. It implies that the way by leveraging the learnt word and design pattern vectors is more appropriate to find a candidate set of design patterns than the way by using the expert classification.

7.3.4 Conclusion

The refined method based on DPWord2Vec is superior to the methods in [54] and [8] on the benchmarks. Therefore, DPWord2Vec contributes to accomplish the task of design pattern selection.

8 Threats to Validity

8.1 Internal validity

There are several threats to internal validity of our work.

First, the size of the corpus may restrict the effectiveness of DPWord2Vec. The corpus in this paper is relatively small comparing with those used in other word embedding methods [11], [13]. This may influence the quality of the learnt word and design pattern vectors. However, we believe this problem would be alleviated as more design pattern relevant documents could be extracted in the coming future due to the popularity of programming forums. Second, only the default values of the parameters are used to build the word and design pattern vectors. However, the empirical study shows that the performance of DPWord2Vec is not very sensitive to the settings of the main parameters, i.e., the context window size for words and the dimension of vectors. Third, the human judgment process of the dp-word pairs may contain uncertainties, since it may be not easy to judge whether a design pattern and a word is related sometimes. However, such procedures are common practice in similarity tasks of various domains [18], [23], [24], [25]. We try to mitigate the uncertainties by involving a new label, i.e., "somewhat related". Moreover, the Fleiss' Kappa measure shows that the annotators reach a substantial agreement. Finally, the way of determining design pattern relevant posts for constructing the crowdsourced corpus is not completely precise. This factor is in the scope of our previous study. We have performed a validation to ensure the reliability of the results [12].

8.2 External Validity

The threats to external validity relate to the generalization of DPWord2Vec. We sample 2,000 dp-word pairs to evaluate DPWord2Vec in terms of dp-word similarity and employ two applications to evaluate DPWord2Vec in terms of design pattern - words (document) similarity. It is unclear whether DPWord2Vec still works well on other tasks. More datasets or applications will be investigated to reduce this threat in the future.

9 Related Work

9.1 Word Embedding for Software Artifacts

Similar to our work, numbers of studies leverage word embedding methods on software artifacts to aid in software engineering relevant tasks.

Some studies focus on mapping APIs into vector space. Nguyen et al. propose API2Vec that learns API vectors based on API usage sequences extracted from code corpora [62]. Similarly, Li et al. embed natural language words and APIs at the same time by leveraging both API sequences and the method comments [40]. To establish API mappings between third-party libraries, Chen et al. present an unsupervised deep learning-based approach to map both API usage semantics and API description semantics into vectors [63].

Meanwhile, some studies aim to learn the representations of programs. Alon et al. produce general representations of programs based on the paths in abstract syntax trees [64]. Henkel et al. represent programs as abstractions of traces obtained from symbolic execution and learn the vectors of the abstractions using word embedding [22]. Piech et al. introduce a neural network method to learn the feature embedding of a whole program and give automatic feedback based on the representation [65].

Moreover, some studies directly use word embedding methods on software-related documents to support some other tasks. Ye et al. train the word embeddings on API relevant documents and aggregate them to estimate semantic similarities between documents [59]. Calefato et al. exploit word embedding on Stack Overflow posts to help to analyse the sentiments of developers [66]. Guo et al. attempt to generate trace links among software artifacts by utilizing word embedding and recurrent neural network trained on clean text from related domain documents [67].

Different from these studies, our work concentrates on associating natural language words and design patterns by embedding them into one vector space. To the best of our knowledge, no previous studies have ever considered the general relatedness between words and design patterns.

9.2 Tag Recommendation in Software Information Sites

In the first application, we apply DPWord2Vec to the design pattern tag recommendation task. There exist a series of tag recommendation methods specified for software information sites.

To automatically recommend tags in software information sites, Xia et al. propose TagCombine which ranks each tag candidate by integrating three ranking component [47]. After that, EnTagRec is proposed and outperforms TagCombine on four software information sites in terms of Recall [48]. To adopt tag recommendation methods in large-scale software information sites, Zhou et al. propose a more scalable approach called TagMulRec [49]. TagMulRec outperforms EnTagRec in terms of Precision and F1-score on four software information sites. Then Wang et al. enhance EnTagRec to a new version, namely EnTagRec++, by leveraging the information of users of software information sites [50]. EnTagRec++ improves TagCombine by over 10% on five software information sites in terms of Recall. Recently, Liu et al. propose FastTagRec which recommends tags using neural network-based classification [51]. An evaluation on ten software information sites shows FastTagRec is more accurate than TagMulRec.

Most of these methods can also be used in the design pattern tag recommendation task. In the evaluation, the DPWord2Vec-based design pattern tag recommendation method is compared against the state-of-the-art ones, i.e., FastTagRec and TagMulRec, to show its effectiveness.

9.3 Design Pattern Selection based on Text

The related work for the second application is about design pattern selection. We focus on the methods leveraging

textual descriptions here. These works can be roughly categorized into two types.

The first type is based on design pattern use cases and recommend design patterns by exploring the most similar use cases to the current design problem. Gomes et al. propose a case-based reasoning approach for design pattern selection and index cases by using WordNet [68]. Similarly, Muangon et al. present a design pattern searching model by combining case-based reasoning and formal concept analysis techniques [10]. Bouassida et al. integrate case search and questionnaire strategy to create an interactive design pattern selection method [69]. These approaches are based on the assumption that there exists a case library. However, few such libraries are publicly available.

The second type is based on general textual descriptions of design patterns. Palma et al. provide an expert system for design pattern recommendation and parses design pattern descriptions to formulate questionnaires for designers [9]. In [70], Pavlič et al. document the knowledge of design patterns by building an ontology for design pattern advisement. The studies [54] and [8] automate the process and only utilize the original descriptions in design pattern books for design pattern selection.

In this application, we follow the automatic design pattern selection framework in [54] and [8] but refine the design pattern class determining phase by DPWord2Vec. The refined method outperforms the methods in [54] and [8] on the benchmarks.

10 Conclusion

In this work, we propose DPWord2Vec, a framework that maps both natural language words and design patterns into one vector space. With the word and design pattern vectors, each design pattern is associated with English natural language. DPWord2Vec leverages the word embedding method to learn the word and design pattern vector representations based on two built corpora with our redefined context windows. An evaluation on a dp-word pair dataset shows that DPWord2Vec is more effective than the baseline methods in measuring the dp-word similarity. Moreover, two design pattern relevant applications are leveraged to investigate the usefulness of DPWord2Vec. The experimental results indicate that DPWord2Vec can outperform the state-of-the-art algorithms on the specific tasks.

In the future, on one hand, we will extract more design pattern relevant documents from other sources to enrich the corpora; on the other hand, we will attempt to apply DPWord2Vec to more design pattern relevant tasks. Moreover, it is also worth investigating the effectiveness of DPWord2Vec on the corpora of non-English languages.

References

[1] Alexander and Christopher, *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[3] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?" *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1213–1231, Sep. 2012.

[4] H. Zhu and I. Bayley, "On the composability of design patterns," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1138–1152, Nov. 2015.

[5] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Detecting the behavior of design patterns through model checking and dynamic analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 4, pp. 1–41, Feb. 2018.

[6] D. Faitelson and S. Tyszberowicz, "Uml diagram refinement (focusing on class and use case diagrams)," in *Proc. of the 39th Int'l. Conf. on Softw. Eng. (ICSE'17)*, 2017, pp. 735–745.

[7] S. Henninger and V. Corrêa, "Software pattern communities: Current practices and challenges," in *Proc. of the 14th Conf. on Patt. Lang. of Programs (PLOP'07)*, 2007, pp. 14:1–14:19.

[8] S. M. H. Hasheminejad and S. Jalili, "Design patterns selection: An automatic two-phase method," *J. Syst. Softw.*, vol. 85, no. 2, pp. 408–424, 2012, special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[9] F. Palma, H. Farzin, Y. Guéhéneuc, and N. Moha, "Recommendation system for design patterns in software development: An dpr overview," in *Proc. of the 3rd Int'l. Workshop on Recomm. Syst. for Softw. Eng. (RSSE'12)*, Jun. 2012, pp. 1–5.

[10] W. Muangon and S. Intakosum, "Case-based reasoning for design patterns searching system," *Int'l. J. Comput. App.*, vol. 70, no. 26, pp. 16–24, May. 2013.

[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. of the 26th Int'l. Conf. on Neural Inform. Proc. Syst. (NIPS'13)*, 2013, pp. 3111–3119.

[12] H. Jiang, D. Liu, X. Chen, H. Liu, and H. Mei, "How are design patterns concerned by developers?" in *Proc. of the 41th Int'l. Conf. on Softw. Eng.: Companion Proc. (ICSE'19 Companion)*, 2019, pp. 232–233.

[13] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. of Empirical Methods in Natural Lang. Proc. (EMNLP'14)*, 2014, pp. 1532–1543.

[14] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proc. of the 39th Intl. Conf. on Softw. Eng. (ICSE'17)*, 2017, pp. 450–461.

[15] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract api mentions from informal natural language discussions," in *2016 IEEE Int'l. Conf. on Softw. Maint. and Evol. (ICSME'16)*, 2016, pp. 389–399.

[16] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proc. of the 19th Intl. Conf. on Program Compreh. (ICPC'11)*, 2011, pp. 11–20.

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

[18] M.-T. Luong, R. Socher, and C. D. Manning, "Better word representations with recursive neural networks for morphology," in *Conf. on Computat. Natural Lang. Learn. (CoNLL'13)*, 2013.

[19] R. Soricut and F. Och, "Unsupervised morphology induction using word embeddings," in *Proc. of the 2015 Conf. of the North American Chapter of the Associat. for Comput. Linguist.: Human Lang. Techn. (NAACL'15)*, May–Jun. 2015, pp. 1627–1637.

[20] X. L. Yang, D. Lo, X. Xia, Z. Y. Wan, and J. L. Sun, "What security questions do developers ask? a large-scale study of stack overflow posts," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 910–924, Sep. 2016.

[21] D. M. W. Powers, "Applications and explanations of zipf's law," in *Proc. of the Joint Conf. on New Methods in Lang. Proc. and Computat. Natural Lang. Learn. (NeMLaP3/CoNLL'98)*, 1998, pp. 151–160.

[22] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proc. of the 26th ACM Joint Meeting on Eur. Softw. Eng. Conf. and Symp. on the Found. of Softw. Eng. (ESEC/FSE'18)*, 2018, pp. 163–174.

[23] E. H. Huang, R. Socher, C. D. Manning, and A. Y. Ng, "Improving word representations via global context and multiple word prototypes," in *Proc. of the 50th Annual Meeting of the Assoc. for Computat. Linguist. (ACL'12)*, 2012, pp. 873–882.

[24] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *2014 IEEE Conf. on Softw. Maint., Reeng. and Reverse Eng. (CSMR-WCRE'14)*, Feb. 2014, pp. 44–53.

[25] A. Mahmoud and G. Bradshaw, "Estimating semantic relatedness in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 10:1–10:35, Dec. 2015.

[26] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Am. Soc. Infor. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.

[27] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1192–1223, Jun. 2016.

[28] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Softw. Eng.*, vol. 19, no. 3, pp. 619–654, Jun. 2014.

[29] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proc. of the 35th Intl. Conf. on Softw. Eng. (ICSE'13)*, May. 2013, pp. 522–531.

[30] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. Natl. Acad. Sci.*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.

[31] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Comput. Linguist.*, vol. 16, no. 1, pp. 22–29, Mar. 1990.

[32] R. L. Cilibrasi and P. M. B. Vitanyi, "The google similarity distance," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 3, pp. 370–383, Mar. 2007.

[33] A. Hotho, A. Nürnberger, and G. Paass, "A brief survey of text mining," *GLDV J. Computat. Linguist. Lang. Technol.*, vol. 20, pp. 19–62, Jan. 2005.

[34] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Trans. Serv. Comput.*, vol. 9, no. 5, pp. 771–783, Sep. 2016.

[35] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proc. of the 36th Int'l. Conf. on Softw. Eng. (ICSE'14)*, 2014, pp. 664–675.

[36] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, Oct. 2013.

[37] Y. Dodge, *The concise encyclopedia of statistics*. Springer, New York, NY, 2010.

[38] S. Siegel, *Non-parametric statistics for the behavioral sciences*. McGraw-Hill, 1956.

[39] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2002.

[40] X. Li, H. Jiang, Y. Kamei, and X. Chen, "Bridging semantic gaps between natural languages and apis with word embedding," *IEEE Trans. Softw. Eng.*, pp. 1–1, 2018.

[41] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. of the 31st Int'l. Conf. on Mach. Learn. (ICML'14)*, 2014, pp. 1188–1196.

[42] C. Zhou, C. Sun, Z. Liu, and F. C. M. Lau, "Category Enhanced Word Embedding," *arXiv e-prints*, p. arXiv:1511.08629, Nov. 2015.

[43] T. Nguyen, N. Tran, H. Phan, T. Nguyen, L. Truong, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Complementing global and local contexts in representing api descriptions to improve api retrieval tasks," in *Proc. of the 26th ACM Joint Meeting on Eur. Softw. Eng. Conf. and Symp. on the Found. of Softw. Eng. (ESEC/FSE'18)*, 2018, pp. 551–562.

[44] J. Cohen, *Statistical power analysis for the behavioral sciences (2nd edition)*. Lawrence Erlbaum Associates, 1988.

[45] S. S. Sawilowsky, "New effect size rules of thumb," *J. Modern Applied Statistical Methods*, vol. 8, no. 2, pp. 597–599, 2009.

[46] J. M. Al-Kofahi, A. Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen, and T. N. Nguyen, "Fuzzy set approach for automatic tagging in evolving software," in *2010 IEEE Int'l. Conf. on Softw. Maint. (ICSM'10)*, Sep. 2010, pp. 1–10.

[47] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proc. of the 10th IEEE Working Conf. on Mining Softw. Reposit. (MSR'13)*, May. 2013, pp. 287–296.

[48] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec: An enhanced tag recommendation system for software information sites," in *2014 IEEE Int'l. Conf. on Softw. Maint. and Evol. (ICSME'14)*, Oct. 2014, pp. 291–300.

[49] P. Zhou, J. Liu, Z. Yang, and G. Zhou, "Scalable tag recommendation for software information sites," in *Proc. of the 24th Int'l. Conf. on Softw. Anal., Evol. and Reeng. (SANER'17)*, Feb. 2017, pp. 272–282.

[50] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec++: An enhanced tag recommendation system for software information sites," *Empirical Softw. Eng.*, vol. 23, no. 2, pp. 800–832, Apr. 2018.

[51] J. Liu, P. Zhou, Z. Yang, X. Liu, and J. Grundy, "Fasttagrec: Fast tag recommendation for software information sites," *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 675–701, Dec. 2018.

[52] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *Proc. of the 21st Nat'l. Conf. on Artif. Intell. (AAAI'06)*, 2006, pp. 775–780.

[53] D.-K. Kim and C. E. Khawand, "An approach to precisely specifying the problem domain of design patterns," *J. Vis. Lang. Comput.*, vol. 18, no. 6, pp. 560–591, 2007.

[54] S. Hussain, J. Keung, and A. A. Khan, "Software design patterns classification and selection using text categorization approach," *Appl. Soft Comput.*, vol. 58, pp. 225–244, 2017.

[55] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, 2001.

[56] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security patterns: integrating security and systems engineering*. John Wiley & Sons, 2006.

[57] B. P. Douglass, *Real-time design patterns: robust scalable architecture for real-time systems*. Addison-Wesley Professional, 2003.

[58] E. M. Voorhees and D. M. Tice, "The trec-8 question answering track report," in *Trec*, vol. 99, 1999, pp. 77–82.

[59] 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 *Proc. of the 38th Intl Conf. on Softw. Eng. (ICSE'16)*, 2016, pp. 404–415.

[60] A. K. Uysal, "An improved global feature selection scheme for text classification," *Expert Syst. App.*, vol. 43, pp. 82–92, 2016.

[61] J. C. Bezdek, *Pattern recognition with fuzzy objective function algorithms*. Springer Science & Business Media, 2013.

[62] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Proc. of the 39th Intl Conf. on Softw. Eng. (ICSE'17)*, May. 2017, pp. 438–449.

[63] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding," *IEEE Trans. Softw. Eng.*, pp. 1–1, 2019.

[64] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proc. of the 39th ACM SIGPLAN Conf. on Programm. Lang. Design and Implement. (PLDI'18)*, 2018, pp. 404–419.

[65] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," in *Proc. of the 32nd Int'l. Conf. on Mach. Learn. (ICML'15)*, vol. 37, Jul. 2015, pp. 1093–1102.

[66] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1352–1382, Jun. 2018.

[67] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. of the 39th Intl. Conf. on Softw. Eng. (ICSE'17)*, May. 2017, pp. 3–14.

[68] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Using cbr for automation of software design patterns," in *Eur. Conf. on Case-Based Reason. (ECCBR'02)*, 2002, pp. 534–548.

[69] N. Bouassida, S. Jamoussi, A. Msaed, and H. Ben-Abdallah, "An interactive design pattern selection method," *J. Universal Comput. Sci.*, vol. 21, no. 13, pp. 1746–1766, Jan. 2015.

[70] L. Pavlič, V. Podgorelec, and M. Hericko, "A question-based design pattern advisement approach," *Comput. Sci. Inform. Syst.*, vol. 11, no. 2, pp. 645–664, Jun. 2014.