Metadata of the chapter that will be visualized in SpringerLink

Book Title	Intelligent Data Engineering and Automated Learning – IDEAL 2020	
Series Title		
Chapter Title	Text Similarity Between Concepts Extracted from Source Code and Documentation	
Copyright Year	2020	
Copyright HolderName	Springer Nature Switzerland AG	
Corresponding Author	Family Name	**Pauzi**
	Particle	
	Given Name	**Zaki**
	Prefix	
	Suffix	
	Role	
	Division	Software and Platform Engineering Chapter
	Organization	BP
	Address	London, UK
	Email	zaki.pauzi@bp.com
Corresponding Author	Family Name	**Capiluppi**
	Particle	
	Given Name	**Andrea**
	Prefix	
	Suffix	
	Role	
	Division	Department of Computer Science
	Organization	University of Groningen
	Address	Groningen, The Netherlands
	Email	a.capiluppi@rug.nl

Abstract

Context: Constant evolution in software systems often results in its documentation losing sync with the content of the source code. The traceability research field has often helped in the past with the aim to recover links between code and documentation, when the two fell out of sync.
Objective: The aim of this paper is to compare the concepts contained within the source code of a system with those extracted from its documentation, in order to detect how similar these two sets are. If vastly different, the difference between the two sets might indicate a considerable ageing of the documentation, and a need to update it.
Methods: In this paper we reduce the source code of 50 software systems to a set of key terms, each containing the concepts of one of the systems sampled. At the same time, we reduce the documentation of each system to another set of key terms. We then use four different approaches for set comparison to detect how the sets are similar.
Results: Using the well known Jaccard index as the benchmark for the comparisons, we have discovered that the cosine distance has excellent comparative powers, and depending on the pre-training of the machine learning model. In particular, the SpaCy and the FastText embeddings offer up to 80% and 90% similarity scores.
Conclusion: For most of the sampled systems, the source code and the documentation tend to contain very similar concepts. Given the accuracy for one pre-trained model (e.g., FastText), it becomes also evident that a few systems show a measurable drift between the concepts contained in the documentation and in the source code.

Keywords Information Retrieval - Text similarity - Natural language processing

Text Similarity Between Concepts Extracted from Source Code and Documentation

Zaki Pauzi[1(✉)] and Andrea Capiluppi[2(✉)]

[1] Software and Platform Engineering Chapter, BP, London, UK
zaki.pauzi@bp.com
[2] Department of Computer Science, University of Groningen, Groningen, The Netherlands
a.capiluppi@rug.nl

Abstract. *Context*: Constant evolution in software systems often results in its documentation losing sync with the content of the source code. The traceability research field has often helped in the past with the aim to recover links between code and documentation, when the two fell out of sync. AQ1

Objective: The aim of this paper is to compare the concepts contained within the source code of a system with those extracted from its documentation, in order to detect how similar these two sets are. If vastly different, the difference between the two sets might indicate a considerable ageing of the documentation, and a need to update it. AQ2

Methods: In this paper we reduce the source code of 50 software systems to a set of key terms, each containing the concepts of one of the systems sampled. At the same time, we reduce the documentation of each system to another set of key terms. We then use four different approaches for set comparison to detect how the sets are similar.

Results: Using the well known Jaccard index as the benchmark for the comparisons, we have discovered that the cosine distance has excellent comparative powers, and depending on the pre-training of the machine learning model. In particular, the SpaCy and the FastText embeddings offer up to 80% and 90% similarity scores.

Conclusion: For most of the sampled systems, the source code and the documentation tend to contain very similar concepts. Given the accuracy for one pre-trained model (e.g., FastText), it becomes also evident that a few systems show a measurable drift between the concepts contained in the documentation and in the source code.

Keywords: Information Retrieval · Text similarity · Natural language processing

1 Introduction

To understand the semantics of a software, we need to comprehend its concepts. Concepts extracted through its keywords from a single software derived from its

C. Analide et al. (Eds.): IDEAL 2020, LNCS 12489, pp. 1–12, 2020.
https://doi.org/10.1007/978-3-030-62362-3_12

source code and documentation have to be consistent to each other and they must be aligned as they define the software's identity, allowing us to classify in accordance to its domain, among others. The importance of similarity stems from the fact that software artefacts should represent a single system where the different cogs are meshed together in synchrony, enabling the moving parts to work together collectively. This also plays a pivotal role in software management and maintenance as software development is usually not a solo effort.

Past research that has attempted this link involved automatic generation of documentation from code such as with SAS [15] and Lambda expressions in Java [1], automatic generation of code from documentation such as with CONCODE [8] and visualising traceability links between the artefacts [7].

Using a sample of 50 open source projects, we extracted the concepts of each project from two sources: the source code and documentation. On one hand, we extracted the concepts emerging from the keywords used in the source code (class and variable names, method signatures, etc) and on the other hand, we extracted the concepts from the plain text description of a system's functionalities, as described in the main documentation file (e.g., the README file). With these two sets of concepts, we run multiple similarity measurements to determine the similitude of these building blocks of what makes a software, a software.

We articulate this paper as follows: in Sect. 3, we describe the methodology that we used to extract the concepts from the source code and the documentation of the sampled systems. In Sect. 4, we illustrate the methods that were implemented to determine the similarity of the two sets of concepts, per system. In Sect. 5, we summarise the results and Sect. 6 discusses the findings of the study. Section 7 finally concludes.

2 Background Research

The study of the traceability links between code and documentation has been conducted for program comprehension and maintenance. As one of the first attempts to discover links between free-text documentation and source code, the work performed in [2] used two Information Retrieval (IR) models on the documentation and programs of one C++ and one Java system. The objective was to uniquely connect one source code item (class, package etc) to a specific documentation page. Our approach is slightly different, and it explores whether there is an overlap between the concepts expressed in the source code and those contained in the documentation.

The field of traceability has become an active research topic: in [12,13] for example, the authors show how traceability links can be obtained between source code and documentation by using the LSI technique (i.e., latent semantic indexing). In our work we show how latent semantic techniques have relatively poor performance as compared to more sophisticated techniques.

In the context of locating topics in source code, the work presented in [10] demonstrated that the Latent Dirichlet Allocation (LDA) technique has a strong applicability in the extraction of topics from classes, packages and overall systems. The LDA technique was used in a later paper [4] to compare how the

topics from source code, and the reading of the main core of documentation, can be used by experts to assign a software system to an application domain. In this paper, however, we combine and enhance past research by using various techniques to extract concepts and keywords from source code, and we compare this set with the concepts and keywords extracted from the documentation.

3 Empirical Approach

In this section, we discuss how we sampled the systems to study and how the two data sources were extracted from the software projects. In summary, we extracted the plain description of software systems, alongside the keywords of their source codes. We then ran a variety of similarity measurements to determine the degree of similarity between the concepts extracted. Figure 1 represents the toolchain visualisation of the approach.

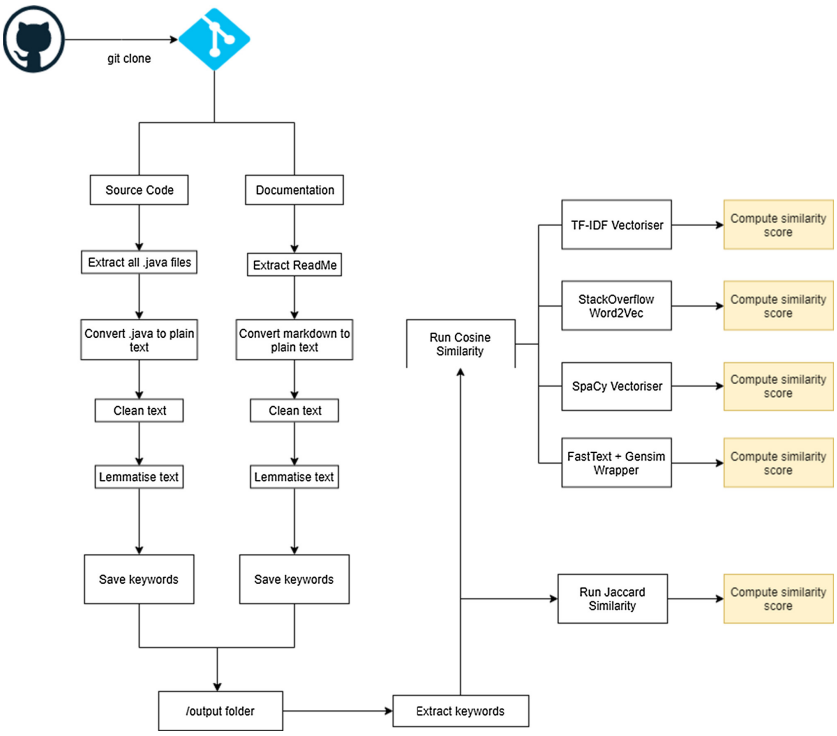

Fig. 1. Toolchain implemented throughout this paper

3.1 Definitions

This section defines the terminology as used throughout the paper. Each of the items described below will be operationalised in one of the subsections of the methodology.

- **Corpus keyword** (term) – given the source code contained in a class, a term is any item that is contained in the source code. We do not consider as a term any of the Java-specific keywords (e.g., `if`, `then`, `switch`, etc.)[1]. Additionally, the camelCase or PascalCase notations are first decoupled in their components (e.g., the class constructor *InvalidRequestTest* produces the terms *invalid*, *request* and *test*).
- **Class corpus** – with the term 'class corpus' we consider the set of all the terms contained within a class. We consider two types of class corpus, per class: the *complete* corpus, with all the terms contained; and the *unique* set of terms, that is, purged of the duplicates.

3.2 Sampling Software Systems and ReadMe Files

Leveraging the GitHub repository, we collected the project IDs of the 50 most successful[2] Java projects hosted on GitHub as case studies. As such, our data set does not represent a random sample, but a complete sub-population based on one attribute (i.e., success) that is related to usage by end users. As a result of the sampling, our selection contains projects that are larger in size than average.

The repository of each project was cloned and stored, and all the Java files (in the latest master branch) identified for further parsing. From each project's folder we extracted the main *ReadMe* file, that is typically assumed to be the first port of information for new users (or developers) of a project.

3.3 Concept Extraction

The extraction was executed in Python, which was then analysed using different text similarity measurements in a Jupyter notebook[3]. This extraction was carried out to build the class corpus for each project. Results were then compared and analysed. For the source codes, we extracted all the class names and identifiers that were used for methods and attributes. These terms are generally written with camel casing in Java and this was handled with the decamelize[4] plugin. Additionally, inline comments were extracted as well. This results in an

[1] The complete list of Java reserved words that we considered is available at https://en.wikipedia.org/wiki/List_of_Java_keywords. The `String` keyword was also considered as a reserved word, and excluded from the text parsing.

[2] As a measure of success, we used the number of *stars* that a project received from other users: that implies appreciation for the quality of the project itself.

[3] Available online at https://github.com/zakipauzi/text-similarity-code-documentation/blob/master/text_similarity.ipynb.

[4] https://github.com/abranhe/decamelize.

extraction that comprehensively represents the semantic overview of concepts whilst minimising noise from code syntax. The final part of the extraction is the lemmatisation of the terms using SpaCy's `token.lemma_`. Lemmatising is simply deriving the root word from the terms, thus enabling more matches when we compare from the different sources.

An excerpt of the complete corpus from the source code of `http-request` is shown at Fig. 2.

```
http request charset content type form content type json encoding
gzip gzip header accept accept header accept charset charset
header ...
```

Fig. 2. Complete class corpus from `http-request` source code (excerpt)

For the documentation, the extraction of concepts for the class corpus from the *ReadMe* markdown file was done by conventional methods such as removing stop words, punctuation and code blocks. All non-English characters were disregarded during the exercise by checking if the character falls within the ASCII Latin space. Figure 3 shows an excerpt of the complete corpus from the documentation of `http-request`.

```
http request simple convenience library use httpurlconnection
make request access response library available mit license usage
httprequest library available maven ...
```

Fig. 3. Complete class corpus from `http-request` documentation (excerpt)

4 Text Similarity

In this section, we address the question of similarity measurements in text. This is crucial to the goal, which is to understand the software's *definition* and be able to apply to it.

Similarity in text can be analysed through lexical, syntactical and semantic similarity methods[6]. Lexical and syntactical similarity depends on the word morphology whereas semantic similarity depends on the underlying meaning of the corpus keywords. The difference between these can be simply understood by the usage of **polysemous** words in different sentences, such as:

1. *"The boy went to the* ***bank*** *to deposit his money."*
2. *"The boy fell by the river* ***bank*** *and dropped his rent deposit money."*

By merely reading these sentences, we understand that they do not mean the same thing or even remotely similar scenarios. This is because we derive meaning by context, such as identifying how the other keywords in the sentence provide meaning to the **polysemous** words. We could also see how the position of *deposit* in the sentences plays a part in differentiating whether it is a noun or a verb. However, a lexical similarity measurement on these two sentences will yield a high degree of similarity given that multiple identical words are being used in both sentences.

These similarity types are a factor in determining the kind of similarity that we are trying to achieve. This is also evident in past research that involved different languages such as cross-lingual textual entailment[16]. In the following sections we describe four different similarity measures that were used to evaluate the two sets of concepts extracted from the software systems.

4.1 Jaccard Similarity

As a measure of similarity, we have used the *Jaccard Similarity* as our baseline measurement. This is one of the oldest similarity indexes [9], that purely focuses on the intersection over union based on the lexical structure of the concepts. This method converts the keywords to sets (thus removing duplicates) and measuring the similarity by the intersection. Figure 4 shows the Venn Diagram acting as a visual representation of the number of keywords in each unique corpus for one of the projects sampled (e.g., the `android-gpuimage` project). Although we could detect 98 *identical* terms in the intersection of the two sets, there are a further 265 terms that are only found in the source code, and some 80 terms that only appear in the documentation set.

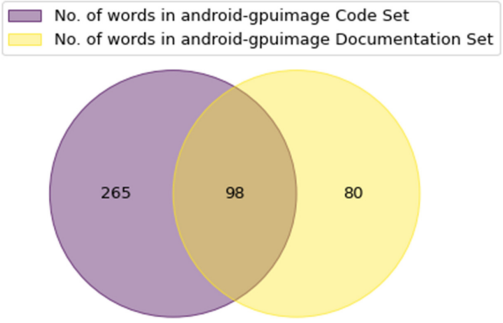

Fig. 4. Intersection of similar concepts by lexical structure

Given how the Jaccard similarity index is being computed, it is important to note that concepts extracted in both sources may not necessarily be written in

an identical manner, but they may be meaning the same. This is reflected by the results, and it becomes apparent when we explore other similarity measurements and compare the results in Sect. 5.

4.2 Cosine Similarity

To handle cases where the keywords are not identical to each other character by character, we have to measure them differently and this is where cosine similarity coupled with word vectors come in. Cosine similarity is a distance metric irrespective of orientation and magnitude. This is particularly important because in a multi-dimensional space, magnitude will not necessarily reflect the degree of similarity, which may be measured using Euclidean distance.

The lower the angle, the higher the similarity. This is measured by:

$$sim(A, B) = cos(\theta) = \frac{A.B}{||A||||B||} = \frac{\sum_1^n A_i B_i}{\sqrt{\sum_1^n A_i^2}\sqrt{\sum_1^n B_i^2}}$$

Another measure of similarity is by the inverse of Word's Mover's Distance (WMD) as first introduced in [11]. This distance measures the *dissimilarity* between two text documents as the minimum amount of distance that the embedded words of one document need to "travel" to reach the embedded words of another document. This method will not be covered in this paper but may be explored in future work.

Term Frequency Inverse Document Frequency (TF-IDF) Vectorizer. Prior to measuring the cosine similarity, we need to embed the terms in a vector space. TF-IDF, short for Term Frequency Inverse Document Frequency, is a statistical measurement to determine how relevant a word is to the document. In this instance, TF-IDF was used to convert the concepts extracted into a Vector Space Model (VSM). Extracting important text features is crucial to the representation of the vector model and this is why we used TF-IDF Vectorizer as term frequency[5] alone will not factor in the inverse importance of commonly used concepts across the documents. As such, the complete corpus was used for this method. TF-IDF for corpus keyword ck is simply the product of the term frequency tf with its inverse document frequency df as follows:

$$ck_(i, j) = tf(i, j) \times log(\frac{N}{df_i})$$

The `TFIDFvectorizer` is provided as a class in the scikit-learn[14] module[6]. The results are shown in Sect. 5.

[5] This can be achieved by the scikit-learn module `CountVectorizer`.

[6] The feature is available in the module named https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TF-IDFVectorizer.

Word Embeddings and Other Vector Space Models. There are numerous word Vector Space Models (VSM) that have been pre-trained such as word embeddings offered by Gensim[7], FastText[8] and SpaCy[9]. We looked into combining the word vectorisation from these modules and then running the cosine similarity measurement to determine the degree of similarity. We have chosen to use these three state-of-the-art word embeddings as they offer a more accurate word representation to compare similarity as opposed to the traditional vectoriser. Terms that were extracted mirror closely to that of the English dictionary of the pre-trained models' vocabulary (StackOverflow, OntoNotes, Wikipedia etc.), making them suitable and effective as word vectorisers.

For Gensim, we used the word2vec word embedding technique based on a pre-trained model on StackOverflow[10] [5] data. As we aim to bring the word vectorisation technique *closest* to home as possible, we decided to use a model that is trained within the software engineering space. On the other hand, as this is a word2vec model, we had to average the results from all the terms extracted as the model is specific for words and not documents. For SpaCy, we used the `en_core_web_md` model, which was trained on OntoNotes[11] and Glove Common Crawl[12].

Finally for FastText, we implemented the vectorisation through the pre-trained `wiki.en.bin` model [3] and loaded it through the Gensim wrapper. This model is trained on Wikipedia data. These vectors in a dimension space of 300 were obtained using the skip-gram model.

5 Results

In this section, we look at how these similarity measurements determine the degree of similarity between terms extracted from source code and documentation. Our baseline measurement, Jaccard Similarity, scores an average of only **7.27%** across the 50 projects. The scores for each project are shown in Fig. 5.

For the cosine similarity measurements, the similarity scores cover a wide range of average results. Table 1 shows the average score for each word vectorisation method combined with cosine similarity. The cosine similarity with FastText Gensim Wrapper Wiki model scored the highest with an average score of **94.21%** across the 50 GitHub projects sampled.

The results for each method are shown in Fig. 6.

[7] https://radimrehurek.com/gensim.
[8] https://fasttext.cc.
[9] https://spacy.io.
[10] https://stackoverflow.com.
[11] https://catalog.ldc.upenn.edu/LDC2013T19.
[12] https://nlp.stanford.edu/projects/glove.

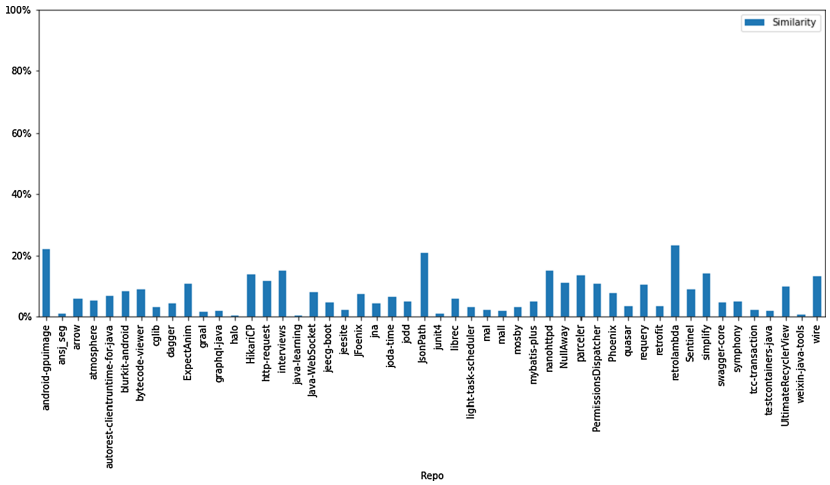

Fig. 5. Jaccard similarity score

Table 1. Average similarity scores across the sampled projects

Method	Score
TFIDFVectorizer	13.38%
StackOverflow Averaged Word2Vec	60.58%
SpaCy en_core_web_md	84.09%
FastText Wiki	94.21%

6 Discussion

It is interesting to note how the degree of similarity can be very different for the various word vectorisation methods. For Jaccard similarity, we can see how the lexical similarity in this scenario does not bode well with concepts extracted from the projects.

For cosine similarity on word vectors, we have a wide range of averages. If we look closely into the trend of projects that consistently score below average among its peers, we can see that this can be attributed to the concepts being non-English, which were disregarded in this paper. This resulted in a lower similarity score across the different measures. When we excluded these projects and only considered those that were fully written in English (42 projects in total), we could see a *noticeable* increase across all measurements. Table 2 shows the new results.

6.1 Limitations

The approach that we have shown has some limitations that we have identified, and we discuss them below:

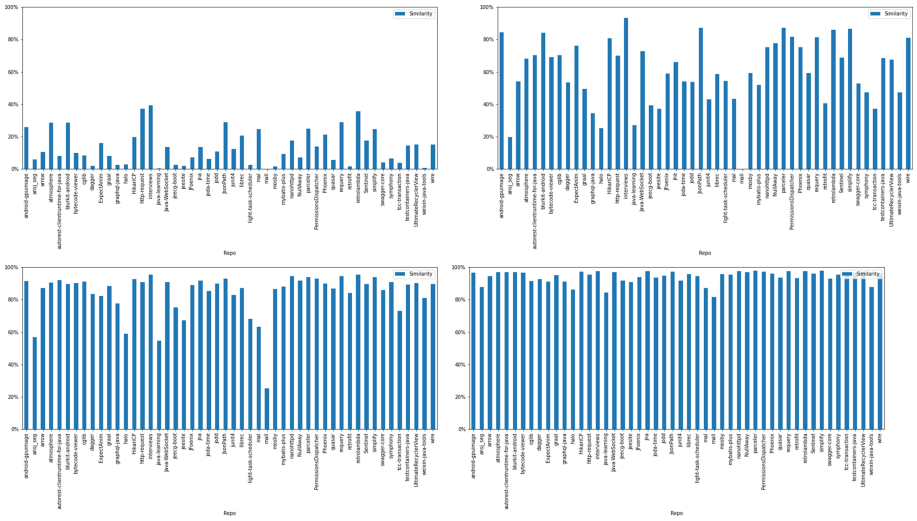

Fig. 6. Cosine similarity scores (the sequence of projects in the X axis is the same as in Fig. 5)

Table 2. Average similarity scores across projects fully in English

Method	Score
TFIDFVectorizer	15.49%
StackOverflow Averaged Word2Vec	66.61%
SpaCy en_core_web_md	88.39%
FastText Wiki	95.36%

1. *Structure of documentation is non-uniform.* Most of the *ReadMe* files are a mixture of code examples and explanations of how to use the system, alongside its options and features. This lack of structure affects the extraction's capability in retrieving the necessary information.
2. *Documentation not in English.* A few of the projects that we analysed are documented in a language different from English (e.g., Chinese). Currently, the extraction of concepts from source code is not affected even in those cases (i.e., the source code is still written with English syntax). On the other hand the non-English documentation cannot be parsed, since our approach ignores non-English characters. This is a limitation because the non-English free text may provide value to the concepts in the code.
3. *Non uniform identifiers.* From the analysis of the systems, we observed that the structure of code has some degree of non-uniformity in the way identifiers are used to represent meaning. In some cases, the classes are explicitly describing some core functionality of the code, while in other cases, they hide the core concepts behind generic descriptors.

7 Conclusion and Further Work

The results of similarity shown between the keywords extracted from the projects' artefacts corroborate on the notion that these terms need to be semantically similar as they represent the constituents of a product. It is important that the terms are similar as they are the core ingredients to software semantics. This research has allowed us to explore this and it has enabled us to move one step forward in applying to the semantics of software.

Bringing this work forward, much can be explored further for the concept extraction and similarity measurement techniques. For example, instead of disregarding the non-English words, we can translate these concepts to English and compare the results. Also, the handling of boilerplate code that does not contribute to the semantics of the software can be further detected and removed. We can also expand our research scope to analyse projects from different languages and whether the structure of the concepts extracted are very much different.

From the NLP standpoint, there has been an increase in research efforts in transfer learning with the introduction of deep pre-trained language models (ELMO, BERT, ULMFIT, Open-GPT, etc.). It is no wonder that deep learning and neural networks will continue to dominate the NLP research field and provide us with an even more effective avenue to further apply to the semantics of software, such as deriving the concepts as a basis of proof for software domain classification, among many others.

References

1. Alqaimi, A., Thongtanunam, P., Treude, C.: Automatically generating documentation for lambda expressions in Java. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 310–320 (2019)
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)
3. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. **5**, 135–146 (2017)
4. Capiluppi, A., et al.: Using the lexicon from source code to determine application domain. In: Proceedings of the Evaluation and Assessment in Software Engineering, pp. 110–119 (2020)
5. Efstathiou, V., Chatzilenas, C., Spinellis, D.: Word embeddings for the software engineering domain. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 38–41 (2018)
6. Ferreira, R., Lins, R.D., Simske, S.J., Freitas, F., Riss, M.: Assessing sentence similarity through lexical, syntactic and semantic analysis. Comput. Speech Lang. **39**, 1–28 (2016). https://doi.org/10.1016/j.csl.2016.01.003, http://www.sciencedirect.com/science/article/pii/S0885230816000048
7. Haefliger, S., Von Krogh, G., Spaeth, S.: Code reuse in open source software. Manag. Sci. **54**(1), 180–193 (2008)
8. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context. CoRR abs/1808.09588 (2018). http://arxiv.org/abs/1808.09588

9. Jaccard, P.: Nouvelles recherches sur la distribution florale. Bull. Soc. Vaud. Sci. Nat. **44**, 223–270 (1908)
10. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. **49**(3), 230–243 (2007)
11. Kusner, M.J., Sun, Y., Kolkin, N.I., Weinberger, K.Q.: From word embeddings to document distances. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML 2015), vol. 37, pp. 957–966. JMLR.org (2015)
12. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 125–135. IEEE (2003)
13. Marcus, A., Maletic, J.I., Sergeyev, A.: Recovery of traceability links between software documentation and source code. Int. J. Softw. Eng. Knowl. Eng. **15**(05), 811–836 (2005)
14. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
15. Righolt, C.H., Monchka, B.A., Mahmud, S.M.: From source code to publication: Code diary, an automatic documentation parser for SAS. SoftwareX **7**, 222–225 (2018). https://doi.org/10.1016/j.softx.2018.07.002, http://www.sciencedirect.com/science/article/pii/S2352711018300669
16. Vilariño, D., Pinto, D., Mireya, T., Leon, S., Castillo, E.: BUAP: lexical and semantic similarity for cross-lingual textual entailment, pp. 706–709, June 2012

Author Queries

Chapter 12

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding authors have been identified as per the information available in the Copyright form.	
AQ2	Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city names in all affiliations. Please check and confirm if the inserted city names are correct. If not, please provide us with the correct city names.	