2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Semantic Source Code Models Using Identifier Embeddings

Vasiliki Efstathiou and Diomidis Spinellis
Athens University of Economics and Business
vefstathiou@aueb.gr, dds@aueb.gr

***Abstract*—The emergence of online open source repositories in the recent years has led to an explosion in the volume of openly available source code, coupled with metadata that relate to a variety of software development activities. As an effect, in line with recent advances in machine learning research, software maintenance activities are switching from symbolic formal methods to data–driven methods. In this context, the rich semantics hidden in source code identifiers provide opportunities for building semantic representations of code which can assist tasks of code search and reuse. To this end, we deliver in the form of pretrained vector space models, distributed code representations for six popular programming languages, namely, Java, Python, PHP, C, C++, and C#. The models are produced using *fastText*, a state–of–the–art library for learning word representations. Each model is trained on data from a single programming language; the code mined for producing all models amounts to over 13.000 repositories. We indicate dissimilarities between natural language and source code, as well as variations in coding conventions in between the different programming languages we processed. We describe how these heterogeneities guided the data preprocessing decisions we took and the selection of the training parameters in the released models. Finally, we propose potential applications of the models and discuss limitations of the models.**

***Index Terms*—fastText, Code Semantics, Vector Space Models, Semantic Similarity**

I. Introduction

The emergence of online open source repositories, such as GitHub, in the recent years has drastically increased the volume of archived software artifacts that are openly available to the community. Such artifacts include source code, combined with an assortment of meta data related to various stages of the development lifecycle. This large–scale mass of data, often referred to as "Big Code" [1] encompasses rich information related to documentation, maintenance events, and authorship of software. An increasing research interest focuses on leveraging the wealth of this data and extracting actionable results for automating related activities.

Data–driven methods have attracted substantial attention, following recent advances in machine learning research and foreseeing the practical potential with the availability of computational resources that can nowadays afford data–intensive tasks. In this context, statistical regularities observed in source code have revealed the repetitive and predictable nature of programming languages, which has been compared to that of natural languages [2], [3]. Consequently, research on problems of automation in natural language processing, such as identification of semantic similarity between texts, translation, text summarisation, word prediction and language generation has inspired parallel lines of research regarding the automation of software development tasks. Relevant problems in software development include clone detection [4], [5], deobfuscation [6], language migration [7], source code summarisation [8], [9], auto–correction [10], [11], auto–completion [12], code generation [13]–[15], and comprehension [16]. The perceived similarity between natural language and source code has largely driven the practice of mining source code, with relevant problems being addressed through the latest state–of–the–art natural language processing methods [6], [8], [15], [17], [18].

Besides similarities, there also exist major differences that need to be taken into consideration when designing such studies. State–of–the art text mining techniques produce impressive results when given sufficient amounts of data expressed in natural language [19]. Substantial volumes of data expressed in a programming language however do not necessarily yield comparable results in equivalent tasks. Especially, when it comes to extracting semantic topics from source code, results are poor. This has been attributed to data sparsity issues [20] as semantically rich elements in source code tend to amount only to a small fragment of the overall data. As a solution to addressing these gaps, we propose the use of pretrained source code embeddings.

The emergence of word embeddings [21] — *i. e.*, representations of words in the continuous vector space — has revolutionized information retrieval in natural language processing. The method relies on the idea that shared textual context implies semantic relatedness, which is in turn reflected as topological proximity in the vector space. At a practical level, this information is delivered through portable models that have been pretrained over large–scale textual data. We claim that on a par with natural language, source code demonstrates similar qualities through the information encoded in source code identifiers. Following the natural language paradigm, we deliver a set of general–purpose models, pretrained over large amounts of code, which can be used to assist a number of information retrieval tasks.

II. Continuous Vector Space Models

Word embeddings are based on the distributional hypothesis proposed by Harris [22], which states that words that occur in the same contexts tend to have similar meanings. Traditional approaches of distributional similarity have treated words as atomic units, represented in a discrete manner as indices in a vocabulary. Sparse, high dimensional vectors for

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00015

encoding this information, however, suffer from scalability issues. Continuous, low dimensional dense vectors provide an alternative representation that overcomes these issues. Continuous representations of words, capture distributional similarity by encoding words into dense vectors where each word is associated with a point in continuous vector space, and semantically related words tend to share context in the vector space. The seminal work by Mikolov et al. [21] with the Word2Vec model brought continuous vector space models into play, with an efficient implementation of an unsupervised algorithm for learning word representations. Follow–up work resulted to the implementation of the *fastText* library [23] by Facebook research which outperforms Word2Vec and, most importantly, builds representations at character-level granularity. This key feature allows the representations of synthetic words that do not appear in the training corpus, and builds models for highly diverse languages that contain many rare words [24].

The success of word embeddings, relies to a certain extent on the fact that pretrained readily available models are easy to access and further exploit by communities with no particular machine learning expertise. Mikolov et al. demonstrated the potential of the method through a Word2Vec model pretrained over 100 billion words of Google news data. The model was released in a portable binary format along with its implementation [25], bringing the method into the mainstream. Similar approaches followed this paradigm [26], releasing toolkits and readily available pretrained models. Ever since, substantial work is constantly under development, oriented towards releasing pretrained general–purpose models [27] in a variety of languages [28], as well as domain–specific embeddings for disambiguating words to their specialized context [29].

In the software engineering community, embeddings have been trained over small datasets pertinent to ad–hoc tasks [30], [31], but also released as general–purpose domain specific–knowledge [32]. In both cases, these models are trained over natural language artifacts related to software development. In this work we provide a collection of models trained on source code in six different programming languages. To the best of our knowledge this is the first set of pretrained source code models to be released for general–purpose use.

III. Source Code Embeddings

Following current trends in natural language processing, with readily available pretrained models being released as exploitable resources of general, common sense knowledge, we propose the release of general–purpose, pre–trained source code models. We motivate towards this idea and describe the implementation steps, from data selection criteria to training the models. We discuss results and demonstrate the potential of the models through a simple code similarity example.

A. Motivation

Good coding practices dictate that source code identifiers be given meaningful, descriptive names. As a result, source code identifiers tend to encompass distinctive semantics that render them useful for communicating information across developers [17]. Furthermore, by considering the fact that code comes in self–contained units of relevant functionality, we postulate that, to a certain extent, contextual distributional semantics in code are captured in ways comparable to those in natural language. The availability of high quality repositories provides the grounds for investigations towards this direction.

B. Data Selection

We selected GitHub public repositories where the primary programming language was one of Java, Python, PHP, C, C++, C#. We chose these languages due to their popularity and diversity in application domains, spanning from web programming to systems programming, and general application programming. In addition, we were interested in training models for languages of varying verbosity, with Python at one end being concise and Java and C# at the other end being more verbose. All six languages are listed within the top 10 most popular programming languages according to Tiobe's index as of January 2019 [33] and are supported by the framework proposed by Munaiah et al. [34] for quality assessment.

We consulted the list of repositories already analyzed by the RepoReapers tool [35], and for each language separately we sorted the related repositories in decreasing order of GitHub stars. We chose repositories with over 100 stars and filtered out of these, few cases of repositories that have not been classified as engineered projects by any of the implemented classifiers [34]. The resulting lists of repositories of a total of $13,144$ repositories that match our selection criteria can be found in our repository [36].

C. Data Collection and Preprocessing

We used a number of shell scripts for compiling and transforming the data in the appropriate format. The complete toolkit is available on our GitHub repository [36]. We followed the preprocessing steps described below.

1) Tokenization: After fetching the selected repositories, we selected source code files with extensions that matched each of the six programming languages of choice, i.e., { .java, .py, .cpp, .php, .c, .cpp, .cs}. We used Tokenizer [37], an open-source tool that provides, among others, functionality for tokenizing source code elements into string tokens. For each programming language we tokenized the content of source code files and stored them in a single file by maintaining their original order. We further preprocessed the tokenized files by filtering out some elements as described in the next section.

2) Data Cleansing: It is a common practice for studies that employ text mining techniques with software artifacts to religiously follow the guidelines akin to natural language processing tasks. Efforts towards adapting to the needs of the task in hand mainly focus on fine–tuning training parameters [20], [38], [39]. The importance of the decisions taken at data preprocessing level is a parameter rarely stressed, despite the fact that the quality of the produced models depends heavily on the features expressed through the representational strength of the data provided. In this study, we performed trials with variations of preprocessed data and decided to follow some

of the standard text mining preprocessing steps and to omit others, as described in our rationale below.

Text Normalization: Lemmatization and stemming are standard normalization techniques employed in order to mitigate the noise produced by grammatical inflections in a variety of natural language processing tasks. However, in continuous vector space models, this type of normalization could lead to information loss as inflections may capture relational analogies, e.g., nominal plural analogies, such as "dog is to dogs what horse is to horses" [40]. Inflection phenomena are not equally pervasive in programming scripts; still source code identifiers do incorporate aspects of inflection, e.g., a class named "Node" versus a collection which is named "nodes" and holds instances of "Node" objects. We maintained the inflected forms of source code identifiers as these originally appear in the scripts. Furthermore, we did not split composite name signatures into their counterparts as we observed that the dictionary of the individual words that compose the highly synthetic vocabulary of a source code document such a class is limited and repetitive. Hence, flattening compositionality of identifiers led to repetition of identical terms limiting the representational strength of the data. Similarly, we maintained typesetting aspects that naming conventions of the different programming languages dictate.

Conversion to Lowercase: Capital case in English language is sparse. Typically words that start with a capital letter are found in the beginning of a sentence, in which case capital case does not assign special semantics to words. The only exception is proper nouns *i.e.*, instances of entities (e.g., "Bob" is an instance of a person). Due to the relative sparsity of named entities, in order to avoid the noise occurring from typesetting diversities it is common practice in text mining to uniformly convert text to lowercase. On the contrary, capital letters used thoroughly in source code text as naming conventions dictate. Naming conventions imply underlying functional semantics of code identifiers. Particularly in object–oriented languages conventions function conversely, with identifiers starting with a capital letter denoting higher–level entities such as classes and interfaces. In order to maintain such features in the data, we maintained the source code text in the original form found in scripts without converting case.

Stop Word Removal: Stop words are short function words that commonly occur in language, and carry limited semantic content (e.g., "the", "and", "this"). Because their presence in a text does not contribute in distinguishing concrete semantics, such words are considered as noise and are often removed at preprocessing in text mining. In analogy to natural language stop words, for each of the programming languages that we mined, we compiled a corresponding list of reserved keywords. The lists of keywords that we filtered out of the data are available in our repository [36].

Punctuation Removal: Heavy use of punctuation is ubiquitous throughout the source code in all six programming languages that we processed, inducing considerable noise. Thus, we decided to remove punctuation symbols in all six languages, except for "_" which is regularly used for compound identifier labels. In addition, we maintained "$" in the PHP dataset due to its use for denoting variables.

Other Noise Removal: We found substantial noise in the data in the form of single characters that occur from a variety of statements (e.g., "e" from "catch Exception e"), numeric values and hexadecimal numbers. We cleared the data from these types of tokens.

The final data set, after cleansing, totals up to nearly one billion tokens. Even though the number is significant, it seems disproportionately low with respect to the overall 2.4 billion lines of code these tokens were obtained from. This implies that a substantial content of the initial data amounts to noise, corroborating the evidence of sparsity of useful information within source code. Table I summarizes key metrics of the data used for training the models.

TABLE I
A SUMMARY OF ANALYZED REPOSITORIES

Pr. Language	# Repos	# Files	# LOC	# Clean Tokens
Java	2,963	2,456,267	589,043,498	258,011,215
Python	3,862	374,225	76,756,824	106,245,311
PHP	2,394	563,258	96,287,040	82,082,221
C	1,826	2,093,090	749,520,681	238,358,382
C++	1,335	2,691,489	822,175,363	167,149,674
C#	764	390,919	69,006,942	92,620,757
Total	13,144	8,569,248	2,402,790,348	944,467,560

D. Training

We used the fastText library [23] for training the models. With fastText word vectors are built from vectors of character substrings contained in a word [24]. This feature allows for the representation of made–up words, hence we found it to be the most appropriate for dealing with artificial languages such as the programming languages under consideration. We used each of the six language–specific consolidated preprocessed files for training the models. We chose the skip-gram model over the cbow-model for training as the former has been observed to be more efficient with subword information [24]. Skip-gram predicts the target word by using a random close-by word within a context window of determined width. We set this to be equal to 5 for all languages besides Python, where we set the context window to be equal to 4 due to the the concise style of the language. For subwords, we set the minimum length of character n-gram to range between 3 and 6. We set the dimensionality of vectors to be equal to 100 and trained the models in 20 epochs. The .bin files of the resulting models are archived on Zenodo. [1] Table II summarizes key metrics of the trained models.

E. Results

In natural language processing, pretrained models are typically evaluated against established benchmarks. Evaluating source code embeddings is not as straightforward. We empirically assessed the models by using the nearest neighbor functionality of *fastText* which, given a query, returns

[1] https://doi.org/10.5281/zenodo.2558730

TABLE II
A SUMMARY OF THE TRAINED MODELS

Pr. Language	Vocabulary Size	.bin File Size (GBs)
Java	2,480,481	2.8
Python	1,005,902	1.6
PHP	715,760	1.4
C	2,734,020	3.1
C++	2,223,393	2.6
C#	990,330	1.6

its closest words in a trained model. We produced several versions of models by changing i) formats of the data, and ii) training parameters. Interestingly, variations in data formatting produced substantial differences with extensively preprocessed data (composite identifier labels split and all tokens lowercased) resulting to poor representations. As discussed in section III-C2, we decided to keep preprocessing minimal for the delivered models. In terms of variations of the training parameters, we experimented with models that ignored subword information and found the produced representations inadequate. Variations in training windows (5–10) and reduction on dimensionality (80–100) of the models did not change the results to our queries dramatically.

In order to obtain a more clear perspective on the value offered by the models and at the same time demonstrate a potential application, we performed a small case study for repository similarity assessment. We used the Word Movers Distance (WMD) [41], a metric proposed for assessing document similarity by considering their embedded word representations in a trained model. We used as documents the tokenized versions of the Java logging libraries SLF4j and Log4j and a similarly–sized general purpose spatial Java library, Spatial4j. By applying pairwise WMD in between the three libraries we found SLF4j and Log4j located closely together with a distance of 0.59 whereas their distances with Spatial4j were equal to 2.39 and 1.99 respectively. Thus, even though the model is trained on a wide range of Java repositories, it incorporates condensed knowledge that renders it capable of drawing out similarity details.

IV. DISCUSSION

The breakout of word embeddings is recent, hence their potential for empowering other processes such as recommendation and classification is currently under development [42], [41], [43], [44]. We propose potential applications of source code embeddings and discuss challenges and limitations that we observed in training and using the models.

A. Opportunities

Combining semantic models of source code together with semantic models of software documentation provides grounds for addressing a variety of problems in software engineering. We mention some indicative examples below.

Identifying Semantic Errors: Semantic errors refer to compilable code which does something other than what is intended to do [3]. Source code embeddings can contribute in inferring semantic inconsistencies and assist tasks such as semantic bug localization and recommendations for semantic bug fix.

Robust Topic Modeling: Agrawal et al. [38] present a comprehensive review of topic modeling studies in software engineering. Following the paradigm of natural language word embeddings, pretrained source code embeddings provide background knowledge that can further enhance existing methods.

Coupling With Other Artifacts: The models can be used for facilitating tasks that require the association of source code with related artifacts in natural language, e.g., assessment of relevance between proposed changes in code and code review comments, recommendation of reviewers, prediction of programming comments.

Auto Completion: The official *fastText* documentation stresses the value of the subword information captured by such models for auto–correction of misspellings. We observed that in a code–writing setting this feature could prove useful for auto-completion with the combined spelling and meaning that identifiers share in the model (e g., identifiers located in the nearest neighborhood of “isFullScreen” include “useFullscreen”, “isFullScreenAllowed”, “toggleFullscreen”, “behindFullscreen”).

B. Challenges and Limitations

The main challenge in producing embeddings for source code identifiers was deciding the appropriate input format for optimizing the representational strength of the models. *fastText* is being used extensively for training word representations for natural languages, there exist however lexical details specific to source code that led to counterintuitive preprocessing decisions as discussed in sections III-C2 and III-E.

In addition to challenges, there are limitations that ensue when models for artificial languages are trained using techniques originally designed for natural languages. Besides their representational strengths, source code embeddings also suffer from weaknesses when compared to their natural language counterparts. *fastText* provides the character n-gram prediction granularity that makes it more suitable for source code than other models, such as Word2Vec. Still, the feature of relational analogy is not prominent in the models. Further work needs to be done at the algorithmic level in order to capture intricacies implicit in source code, which do not apply in the case of natural language. Research towards this direction is just being initiated [45]. In the meantime, we believe that semantic representations of source code, embedded in dense vector space models, can drive a variety of information retrieval tasks in software engineering.

ACKNOWLEDGMENT

The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union’s Horizon 2020 Research and Innovation Programme under grant agreement No. 732223 and as part of the FASTEN project which has received funding from the European Union’s Horizon 2020 Research and Innovation Programme under grant agreement No. 825328.

REFERENCES

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012.

[3] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[4] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016.

[5] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017.

[6] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 683–693.

[7] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013.

[8] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2016, pp. 2073–2083.

[9] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016.

[10] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016.

[11] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning." in *AAAI*, 2017.

[12] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012.

[13] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015.

[14] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2016.

[15] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2017.

[16] C. V. Alexandru, S. Panichella, and H. C. Gall, "Replicating parser behavior using neural machine translation," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017.

[17] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013.

[18] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for Smell Detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, Universita di Salerno, Salerno, Italy. IEEE, 2016.

[19] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, 2009.

[20] A. Mahmoud and G. Bradshaw, "Semantic topic models for source code analysis," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1965–2000, 2017.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

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

[23] "fasttext - library for efficient text classification and representation learning," https://fasttext.cc/, [last accessed 5-Feb-2019].

[24] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[25] "word2vec - tool for computing continuous distributed representations of words." https://code.google.com/archive/p/word2vec/, [last accessed 5-Feb-2019].

[26] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[27] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, "Advances in pre-training distributed word representations," in *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[28] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov, "Learning word vectors for 157 languages," in *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[29] K. Taghipour and H. T. Ng, "Semi-supervised word sense disambiguation using word embeddings in general and specific domains," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2015.

[30] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016.

[31] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.

[32] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018.

[33] "Tiobe - the software quality company," https://www.tiobe.com/tiobe-index/, 2016, [last accessed 5-Feb-2019].

[34] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[35] "Home of the reporeapers," https://reporeapers.github.io/results/1.html, [last accessed 5-Feb-2019].

[36] https://github.com/vefstathiou/scode-ft-embeddings, 2019, [last accessed 5-Feb-2019].

[37] D. Spinellis, "dspinellis/tokenizer: Version 1.1," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2558420

[38] A. Agrawal, W. Fu, and T. Menzies, "What is wrong with topic modeling?(and how to fix it using search-based software engineering)," *Empirical Software Engineering*, vol. 4, p. 2, 2018.

[39] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[40] G. Finley, S. Farmer, and S. Pakhomov, "What analogies reveal about word vectors and their compositionality," in *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (* SEM 2017)*, 2017.

[41] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *International Conference on Machine Learning*, 2015.

[42] O. Barkan and N. Koenigstein, "Item2vec: neural item embedding for collaborative filtering," in *Machine Learning for Signal Processing (MLSP), 2016 IEEE 26th International Workshop on*. IEEE, 2016.

[43] R. Fu, J. Guo, B. Qin, W. Che, H. Wang, and T. Liu, "Learning semantic hierarchies via word embeddings," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, vol. 1, 2014.

[44] N. Rekabsaz, B. Mitra, M. Lupu, and A. Hanbury, "Toward incorporation of relevant documents in word2vec," *arXiv preprint arXiv:1707.06598*, 2017.

[45] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019.