Feature Extraction from JavaScript

Tamás Aladics, Judit Jász and Rudolf Ferenc

Abstract: Source code analyzation is generally a challenging task and it is especially true for loosely typed languages like JavaScript. Traditionally analyzation is done by hand with the help of static analyzation tools which has many disadvantages - one of which is the lack of robustness. The recent advances in machine learning are promising to increase the robustness of source code analysis, however for ML models to work a meaningful and compatible representation is needed. We propose a specific way of extracting features of JavaScript source code based on it's underlying structure (AST) then we embed these features to a fixed length vector using Doc2Vec. Applying this method on a dataset of 150 000 Java Script source files we found this representation to be meaningful as the semantically similar AST nodes are grouped together after the embedding.

Keywords: JavaScript, Feature Assembler, Deep Learning, Doc2Vec

Introduction

Machine learning and recently deep learning has set foot in most subfields of computer science and source code analysis is no exception. Machine learning can be used by itself or as a supplementary tool for various tasks in this domain, for example code summarization, code completion and error detection. For this purpose it is important to find meaningful representation and attributes of the source code that can be fed to machine learning models as features.

This is a challenging task for various reasons. For example the lately highly successful natural language processing methods to process inputs are optimized for natural languages leaving the domain of artifical languages with potentially better working approaches. Another challenge is the fact that different programming languages may be better represented in different ways as they are highly diverse in their sytax and semantics. Loosely typed languages are especially challenging in this regard.

In this work we propose a method to extract meaningful features from source code focusing on JavaScript methods and functions. We took advantage of the strictly structured nature of programming languages and used source code's underlying tree repesentation (Abstract Syntax Tree or AST) to derive features. This involved embedding each function's AST to variable length vectors in ways that try to preserve the graph's structure.

Using this method to produce the input sentences we trained a Doc2Vec embedding [1] on a database containing 150 000 JavaScript source files [2]. In the end we find that meaningful representation is indeed apparent as in the resulting embedding space the semantically related nodes are close to each other.

Related

In the field of source code analysis different methods have been proposed to find ways to represent source code digestible by machine learning models. One way to categorize these approaches is their granuality, as it is done by Zimin Chen et al. [3] which we use to present the related work in this topic.

Various attempts have been made to extract features based on the tokens that build up the source code. Harer et al. [4] uses these tokens as inputs for Word2Vec to generate word embedding for C/C++ tokens for software vulnerability prediction. Chen & Monperrus [5] use Word2Vec to generate java token embedding for finding the correct ingredient in automated program repair . They use cosine similarity on the embeddings to compute a distance between pieces of code.

Other approaches are based on embedding functions or methods, which is not as fine grained as representations based on tokens. Devlin et al. [6] use function embedding as input to repair variable misuse in Python. They encode the function AST by doing a depth first traversal and create an embedding by concatenating the absolute position of the node, the type of node, the relationship between the node and its parent and the string label of the node. DeFreez et al. [7] generate function embeddings for C code using control-flow graphs. They perform a random walk on interprocedural paths in the program, and used the paths to generate function embeddings. The embedding is used for detecting function clones.

In the realm of JavaScript source analyzation a related work is done by Theeten et al. [8] who presented Import2Vec, an embedding of software libraries for (among other languages) JavaScript. They create vectors based on import statements to indicate similarity between software libraries. A lot of other methods to tackle source code analysis are based on metrics. Gregor Richards et al. [9] uses various (both general and dynamic language related) metrics to analyse the dynamic behaviour of JavaScript.

Taking the work of Zimin Chen et al. [3] as basis, numerous papers have been published in the field of feature extraction but only a small subset of these use AST as we propose. Also, most of the publications are focused around Java, C/C++ or Python; the publications involving JavaScript are very limited in quantity. This small number of publication for JavaScript are solely based on metrics and static analysis tools [9, 10], or they are problem specific [8]. Our method tries to specialize for JavaScript (as we took extra steps to take care of the heavy use of anonymous functions, which is not that elemental in the more object oriented languages) but still remain general.

Methodology and results

The first step of deriving important features of source code was to get the underlying structure of it. For this we used SourceMeter, an open source static analysis framework [11]. Analyzing with SourceMeter generates various results and one of them is an AST representation which we used. However, AST in itself is not a compatible format for machine learning algorithms as it is a graph.

To map each AST to a form that is usable by ML algorithms we flattened them to AST node sequences with the use of SourceMeter tools. Constructing the sequence consists of traversing the AST, getting each node's kind (ie. identifier, assignment, for loop etc.) then outputting these types. Furthermore, we needed to find a way to take the overall structure of the code into account so that *if(cond) exp1; expr2;* and *if(cond) expr1; expr2;* will not be mapped to the same node sequence. For this end we introduced a separate identifier that is inserted into the node sequence everytime the scope is changed. With this extra step we tried to ensure that the resulting vector will be specific enough to reflect the structure but still remain general and simple. Another note be added that JavaScript has heavy use of nested functions. We took the path of only flattening the most outer function, the inner functions are flattened in place and are not added as separate node sequences.

After flattening the AST we acquired a node sequence, each element in the sequence is an identifier of a node kind. However, these sequences hold no semantic information yet. We used Doc2Vec [1] to map these node sequences into meaningful representations in form of vectors. Doc2Vec in general is an extension of Word2Vec [12], and it's purpose is to generate vectors for documents and for the words that build the documents up, using a specific mapping. This mapping's objective is to map semantically similar words and documents to vectors whose distance is minimal for words/documents that are semantically similar. There are two main approaches for this mapping: PV-DM and PV-DBOW. Both algorithms use a sliding windows (context) around each word (center), and a paragraph vector for each document. The difference is in the way of vector generation: PV-DM adjusts vectors so that context words and the

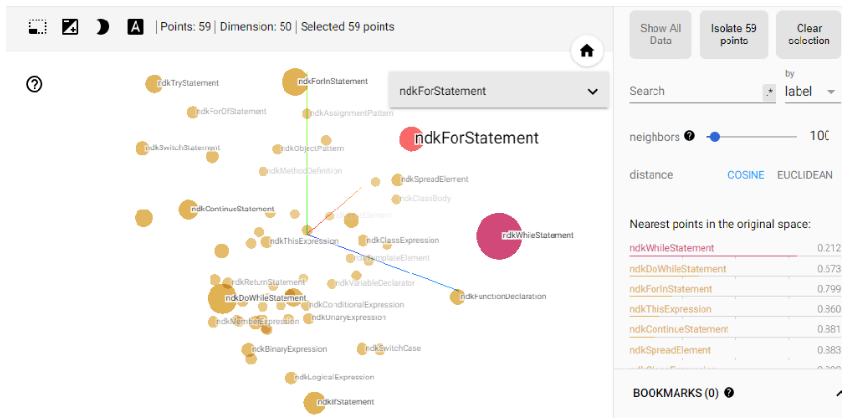

Figure 1: For Statement in the dimension reduced embedding space

paragraph vector predict the center word. PV-DBOW tries to do the opposite in a generative manner: it calibrates vectors so that the central word generates the context words and the paragraph vector. In the end both PV-DM and PV-DBOW finds an embedding where word vectors will hold semantic information on word level, and document (or paragraph) vectors will hold information on document level.

The embedding must be trained on a corpus that is large enough. We used a dataset made up of 150 000 JavaScript source codes from more than 9300 projects [2]. This codebase is promising to be diverse enough: it contains projects that consist of only a few configuration files to complex systems. Using SourceMeter we generated the ASTs for the whole code base, then on each AST we ran our AST flattener which resulted in 597 074 functions flattened into node kind sequenes.

In our work, using the Doc2Vec lingo the "documents" are the node kind sequences (that corresponds to a function), and the "words" are the node kinds in them. We used the genism [13] library's PV-DM algorithm implementation to generate the embedding, using the vector size of 50, windows size of 10 with 8 epochs over the 597074 functions acquired from the 150k JavaScript dataset.

We found that using the described AST Flattening on a function level then utilizing Doc2Vec to get the embedding for the vectors resulted in a meaningful representation of the source code. To illustrate these results we used Tensorboard, Tensorflow's profiler and visualization toolkit, which has built-in ways to present higher dimension vectors in 3D, in our case we used PCA to do the dimensionality reduction. For exmaple, on Figure 1 we can see that the node kind corresponding to *for loops* is most similar to loops and other control flow statements.

Conclusions and future works

To concolude our work it is apparent that this way of representing JavaScipt source code can be promising as it preserves semantic information to an extent. However for it to be truly usable as part of an ML model it must be modified for the specific task: a database with fine tuned features and labels and different parameters can be tried for the Doc2Vec embedding to find the most optimal settings.

Some modification in the AST Flattener may also be benefical based on the task, for example the handling of nested function methods. In our work we flattened the inner methods in place, however using references and adding them as separate instances of methods could prove useful as this could potentially increase the number of available function methods at the cost of losing some structural information.

Another possible modification in future works could be to use different ways to learn the

embeddings than Doc2Vec. Graph2Vec for example is a good candidate as it may be better at capturing the behaviour of the AST as it is a special graph.

Acknowledgements

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004)[1].

References

[1] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[2] ETH Zürich. 150k Javascript Dataset. https://www.sri.inf.ethz.ch/js150.

[3] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *CoRR*, abs/1904.03061, 2019.

[4] Jacob Harer, Louis Kim, Rebecca Russell, Onur Ozdemir, Leonard Kosta, Akshay Rangamani, Lei Hamilton, Gabriel Centeno, Jonathan Key, Paul Ellingwood, Marc McConley, Jeffrey Opper, Sang Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. 02 2018.

[5] Zimin Chen and Martin Monperrus. The remarkable role of similarity in redundancy-based program repair. *CoRR*, abs/1811.05703, 2018.

[6] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *CoRR*, abs/1710.11054, 2017.

[7] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. Path-based function embedding and its application to specification mining. *CoRR*, abs/1802.07779, 2018.

[8] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec - learning embeddings for software libraries. *CoRR*, abs/1904.03990, 2019.

[9] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. volume 45, pages 1–12, 05 2010.

[10] Wei-Hong WANG, Yin-Jun LV, Hui-Bing CHEN, and Zhao-Lin FANG. A static malicious javascript detection using svm. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013/03.

[11] FrontEndArt Software Ltd. Sourcemeter. https://www.sourcemeter.com/.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[13] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[1]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.