Learning Semantic Vector Representations of Source Code via a Siamese Neural Network

David Wehr*, Halley Fede†, Eleanor Pence‡, Bo Zhang§, Guilherme Ferreira§, John Walczyk§ and Joseph Hughes§
*Iowa State University, Ames, IA, USA
dawehr@iastate.edu
†Rensselaer Polytechnic Institute, Troy, NY, USA
halleyfede@gmail.com
‡Massachusetts Institute of Technology, Cambridge, MA, USA
eleanorp@mit.edu
§IBM, Durham, NC, USA
{bozhang, grferrei, jwalczyk, jahughes}@us.ibm.com

***Abstract*—The abundance of open-source code, coupled with the success of recent advances in deep learning for natural language processing, has given rise to a promising new application of machine learning to source code. In this work, we explore the use of a Siamese recurrent neural network model on Python source code to create vectors which capture the semantics of code. We evaluate the quality of embeddings by identifying which problem from a programming competition the code solves. Our model significantly outperforms a bag-of-tokens embedding, providing promising results for improving code embeddings that can be used in future software engineering tasks.**

I. Introduction

Research shows that up to 20% of code within large software projects is actually duplicated code [1] — either identical, "copy & pasted" duplicates, or, more commonly, copied and then further modified to suit a specific need. Within an organization, duplication can also occur from developers recreating similar libraries and tools. All of these duplicates have to be separately debugged, reviewed, and maintained by future developers, wasting time and creating more potential locations for introducing bugs.

Code duplication, known as software or code clones, is a deeply studied topic in the field of software analysis, given the potential benefits of accurately identifying them. A report by Roy and Cordy [1] covers the state-of-the-art as of 2007 and defines different types of code clones, ranging from exact duplicates (Type I) to functionally similar but syntactically different pieces of code (Type IV). At the time of the report, hardly any attempts had been made at identifying Type IV clones.

Since 2007, though, machine learning, and specifically deep learning, have revolutionized many fields, such as computer vision, natural language processing (NLP), business, medical, and games [2] [3]. Given the importance of software development today, it's worth exploring applying these methodologies to software analysis and clone detection to see if similar improvements could be made. Many machine learning applications require transforming the code input to a vector representation, where vectors close to each other in space represent inputs which are similar [4]. For most applications, the vectors should represent semantics (meaning), rather than just the syntax and structure. For example, the same program can be implemented in different ways, and we would like the vectors for different implementations to be categorized as the same.

In this paper, we propose a training method for deep learning models using sets of code snippets known to implement a particular function. These code snippets can be obtained from programming competitions or synthesized via source code modification. We use a Siamese neural network to produce embeddings such that distances represent semantic similarity.

Our main contributions in this paper are as follows:

- Our model learns vectors that summarize code by utilizing the similarity among code snippets. These general vectors can then be used for various applications, including duplicate detection, bug detection, etc.
- Our method leverages verified semantically equivalent code to directly learn the equivalency between two pieces of code, even if their abstract syntax trees (AST) have different structures.
- The proposed model can make use of large sets of unlabeled source code as part of a pre-training stage, which reduces the need for difficult-to-obtain labeled training sets.

The rest of this paper is organized as follows. Section II describes the related work. Section III describes the detailed design of our approach. Section IV presents the training data and evaluation results. We conclude the paper in Section V.

II. Related Work

There have been several studies on code representation and modeling. However, they are often developed with a specific task in mind, rather than attempting to create general code representations. Our approach is unique because it relies on less assumptions with regards to the data, and it is applicable to recognize the semantic equivalency for any kind of code structures. We explain similar and supporting work below.

Gu et al. introduce CODEnn [5], which creates vector representations of Java source code by jointly embedding code with a natural language description of the method. Their architecture uses recurrent neural networks (RNN) on sequences of API calls and on tokens from the method name. It then fuses this with the output of a multi-layer perceptron which takes inputs from the non-API tokens in the code. By jointly embedding the code with natural language, the learned vectors are tailored to summarize code at a human-level description, which may not always be accurate (given that code often evolves independently from comments), and is limited by the ability of natural langauges to describe specifications. Additionally, natural languages (and consequently, code comments) are context-sensitive, so the comment may be missing crucial information about the semantics of the code.

White et al. [6] convert the AST into a binary tree and then use an autoencoder to learn an embedding model for each node. This learned embedding model is applied recursively to the tree to obtain a final embedding for the root node. The model is an autoencoder, and as such, may fail to recognize the semantic equivalency between different implementations of the same algorithm. e.g. that a "for" loop and a "while" loop are equivalent.

The work of Mout et al. [7] encodes nodes of an AST using a weighted mixture of left and right weight matrices and children of that node. A tree-based convolutional neural network (CNN) is then applied against the tree to encode the AST. The only way semantic equivalents are learned in this model are by recognizing that certain nodes have the same children. This assumption is not necessarily correct, and as such, may not fully capture the semantic meaning of code.

Dam et al. [8] used a long-short term memory (LSTM) cell in a tree structure applied to an AST to classify defects. The model is trained in an unsupervised manner to predict a node from its children. Alon et al. [9] learn code vector embeddings by evaluating paths in the AST and evaluate the resulting vectors by predicting method names from code snippets. Saini et al. [10] focus on identifying clones that fall between Type III and Type IV by using a deep neural network — their model is limited by only using 24 method summary metrics as input, and so cannot deeply evaluate the code itself.

In Deep Code Comment Generation [11], Hu et al. introduce structure-based traversals of an AST to feed into a sequence to sequence architecture, and train the model to translate source code into comments. Then the trained model is used to generate comments on new source code. The encoded vectors are designed to initialize a decoding comment generation phase, rather than be used directly, so they are not necessarily smooth, nor suitable for interpretation as semantic meaning. Nevertheless, we draw inspiration from their work for our model.

III. Proposed Model

We would like to process code using NLP techniques — one difficulty in this is that the internal structure of source code is the abstract syntax tree, while most NLP techniques are designed to work on a linear sequence of words. Therefore, we need to flatten the AST into a sequence. A method proposed by by Hu et. al [11] is called structure-based traversal (SBT). One strength of SBT for our application is that since it is fundamentally a depth-first traversal, subtrees are described by consecutive tokens, which often represent a discrete piece of functionality (e.g. encapsulated by an "if" statement). SBT also adds delimiters at the start and end of a subtree, which makes the traversal unambiguous. The delimiters are especially useful for recurrent neural networks with an internal state and gates (e.g. LSTM and GRU units), as they provide an explicit signal for when to trigger the gates.

As is typical in NLP, we limit our token vocabulary to the most common tokens, replacing the others with a generic "<UNK>" token. Although [11] argue that such a representation is inappropriate because of the wide variety in tokens, we found that a small vocabulary size can cover the majority of tokens encountered (Section IV-A).

With the source code transformed to a sequence of tokens, it is suitable to feed into various NLP architectures. Figure 1 shows a high-level overview of our model. The general structure is that of a Siamese neural network [12]. In it, two samples are fed forward through the model, producing two vectors v_1 and v_2. The distance between the vectors is computed, and the loss function is designed so as to make the distance inversely proportional to the semantic similarity of the code. For our data, we only know if two pieces of code belong to the same class (implement the same functionality) or not, so a simple loss function, described by Sun et. al. [13] is shown in eqn. 1, where $y = 1$ if the two vectors belong to the same class, and $y = 0$ otherwise. s is the cosine similarity (eqn. 2), which we chose because of the high dimensionality of our embedding vectors.

$$l(s, y) = \frac{1}{2}\left(y - \sigma(ws + b)\right)^2 \tag{1}$$

$$s = \frac{v_1 \cdot v_2}{\|v_1\| \, \|v_2\|} \tag{2}$$

Parameters w and b in eqn. 1 are trainable scalar values which scale and shift the similarity value.

More complex loss functions for Siamese neural networks have been developed, many of which enforce only a margin between classes [14], and some which use a triplet of (anchor, positive sample, negative sample) [15]. These losses often result in smoother embeddings and faster convergence, but exploring them for code representation remains as future work.

To transform the SBT sequence into a vector that can be used in the loss function, we chose to use a single-layer RNN with a 128-dimensional LSTM cell. We ignore the output of the LSTM cell, and use the final hidden state as the vector embedding of the code. Other network architectures that operate on sequences could also be used, such as the Transformer [16], which has shown promising results on other language tasks.

To summarize, our model uses an LSTM RNN to encode the SBT representation of an AST. It is then trained on labeled

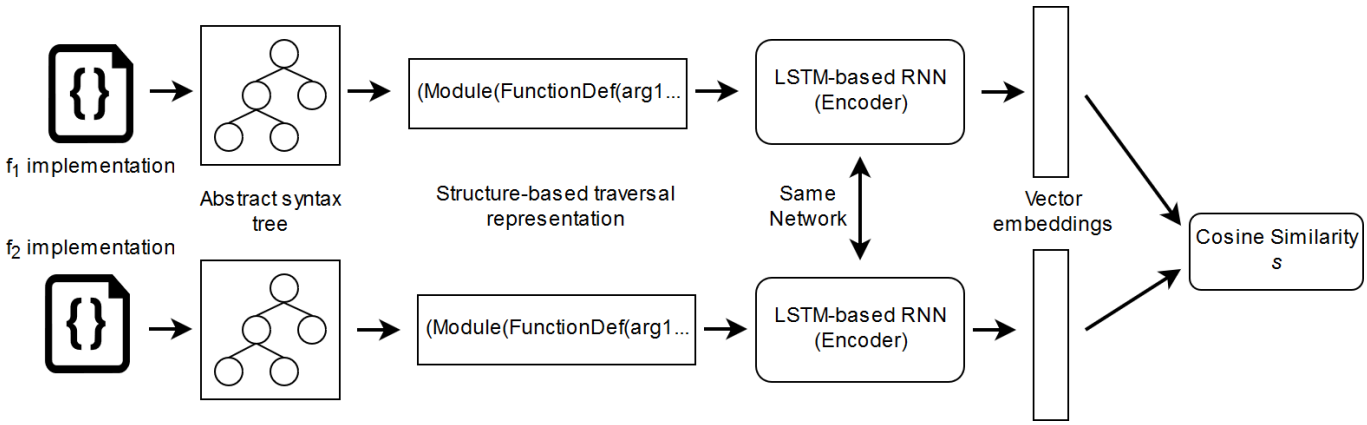

Fig. 1: Structure of Siamese neural network for learning vector embeddings. f_1 and f_2 may be different or the same functions. Also note that both the upper and lower branches of the network use the same structure and weights for the recurrent neural network embedding network.

data where each sample implements a function $f \in F$, where F is a set of unique semantic classes (see Sec. IV on how we obtain these). Two samples are fed into the forward pass of the network, resulting in vectors v_1 and v_2. The loss function penalizes low distances when $f_1 \neq f_2$, and high distances when $f_1 = f_2$. From this, the model learns to produce vectors that represent the semantic functionality of code.

A. Pre-training

Because labeled training data is hard to obtain (requiring independent implementations), we pre-train the network by adding a decoder RNN after the encoding stage, turning it into a sequence-to-sequence model [17]. This is trained as an autoencoder, with the objective of generating the input SBT during the decoding stage. The decoder stage is then discarded for training the supervised Siamese model. This unsupervised pre-training allows the model to consume a large amount of unlabeled data, thereby learning initial weights of the encoder stage that help to parse SBTs. Both the pre-training and final model were trained using the Adam [18] weight-update algorithm using standard backpropagation techniques.

IV. Empirical Analysis

The goal of our evaluation is to determine whether our model can detect when snippets of code perform the same function regardless of implementation. These are known as Type-IV code clones, and represent the most difficult class of duplicate detection. Code competition platforms represent a good source of type-IV clones, because implementations are independently written by different authors, yet each implementation passes the same automated tests for the problem.

A. Data

For pre-training, we collected all Python projects on public GitHub [19] that had over 100 stars (using code from [20]) and extracted their functions as separate training samples, resulting in 1.3 million total samples. Our vocabulary was defined by using the most common tokens in this GitHub data, with a cutoff such that 85% of the encountered tokens in the code were included in our vocabulary, resulting in 1772 total tokens. Many of the same tokens are used in different projects (e.g. Python standard library function names, common libraries, etc.), so a small number of vocabulary tokens are able to cover a large proportion of all code.

For the Siamese neural network data, we scraped data from HackerRank [21] giving us the following data:

1) A set (S_1) of 44 different challenges, split into $S_1^{(train)}$ and $S_1^{(test)}$, each with 4000 and 1000 samples per challenge, respectively.
2) A set (S_2) of 42 different challenges, with each challenge having between 3 and 5000 solutions, for a total of $\sim$42 000 samples. The median number of solutions is 283.

$S_1^{(train)}$ was used for training the Siamese network. These two separate problem sets allow us to evaluate how the model performs on unseen implementations ($S_1^{(test)}$) as well as its generalization to new problems (S_2).

The pre-training stage of our model worked exclusively with function definitions, so to make the Siamese network data consistent with that, the AST of each solution file was wrapped in a function definition. This syntax manipulation is possible in Python, but may not work for all languages.

B. Classification Evaluation

Our goal is to create embedding vectors that allow the detection of code duplicates. But for any classifier, there is a trade-off between false positives and false negatives. To capture the performance of the embeddings independently of this trade-off, we calculate the performance at different distance thresholds and plot the result into a receiver operating characteristic (ROC) curve. The area under the curve (AUC) serves as an aggregate measure of the model performance.

Although different implementations that solve the same problem are unlikely to be identical, they may have many common syntactical features. For example, if the input data is provided in a list of lists, many solutions will have a nested

	$S_1^{(test)}$	S_2
Proposed model	0.98	0.89
Baseline model	0.82	0.76

TABLE I: AUC scores by dataset and model

loop to read the data. To verify that our model is producing embeddings beyond the syntactical level, we compare it against a simple model that just uses syntax. The naive "baseline" model we compare it against is a bag-of-words embedding that represents the frequency of each token in a 1772-dimension vector and compare it against our model. You can also view the baseline vectors as a histogram of token occurrence.

Figure 2 shows the ROC curves for both the proposed model and the baseline model, for both data sets $S_1^{(test)}$ and S_2. The AUC metrics are summarized in Table I.

C. Error Analysis

To gain a sense of how the errors are distributed, we plotted a grid showing the cosine distances between every vector pair for each challenge with over 200 code samples in S_2 (Figure 3). The embedding vectors are grouped by challenge, so there are 26 bands, where each band has 200 randomly sampled solutions to the challenge. Ideally, each of the 26 square groups along the diagonal would have a distance of 0 (yellow), with all other parts of the grid having distance 1 (blue). Any light colors outside of the diagonal squares indicates potential errors, depending upon the classification threshold.

The resulting image shows that for a given challenge (row), most errors are confined to just a few challenges that have similar embedding vectors This is in contrast to having the errors evenly distributed among all challenges. The observed error distribution indicates that some challenge solutions looked identical to our model, which is likely due to them differing in ways which were not observed in the training data. This provides support that a wider selection of training challenges could improve the results.

D. Discussion

As is to be expected, our model shows an extremely strong ability to classify samples from the problems it was trained on (S_1), but in addition, with an AUC of 0.89, the ability to generalize to new, unseen problems (S_2) is also fairly strong. The performance on unseen problems is especially promising given the relatively low number of unique problems it was trained on (44) — it should be expected that training on a larger selection of problems will improve the embeddings further.

Due to our decision to test our model on code written in Python, it is difficult to compare our results with other work in this field, which typically focuses on Java, and we also cannot evaluate against standard datasets like BigCloneBench [22]. We hope that our use of publicly-obtained data (online programming competition) can serve as a repeatable evaluation for future research.

V. Conclusion and Future Work

This paper investigates a deep-learning based approach for code duplicate detection. The proposed model uses a RNN-based Siamese neural network for generating code vector representations that represent semantic similarity between codes. The model also makes use of large amounts of easily-obtained source code in a pre-training autoencoder phase. We evaluate the embedding quality by considering solutions to coding competition questions as functionally identical. Evaluations show that our model can identify whether two pieces of code implement the same functionality with substantially higher accuracy than a naive syntax-based model. This work can serve as a foundation for improving the many other use cases of semantic code embeddings, such as bug detection, code recommendation, and code search.

One current challenge in neural network-based code analysis is how to deal with out-of-vocabulary tokens. Although we were able to cover 85% of source code with fewer than 2000 tokens, encoding the remaining 15% could improve the results further. One approach to mitigate this [11] is to use the variable type, but because Python does not declare types, using inferred types, such as from mypy [23], may work as a substitute. Other future work includes exploring more complex loss functions for the Siamese network, training on larger datasets, and applying the model to other languages, such as Java.

Acknowledgment

Special thanks to the IBM Extreme Blue program. We would also like to express our gratitude to Ross Grady, RTP Lab Manager for IBM Extreme Blue, for his patience and support.

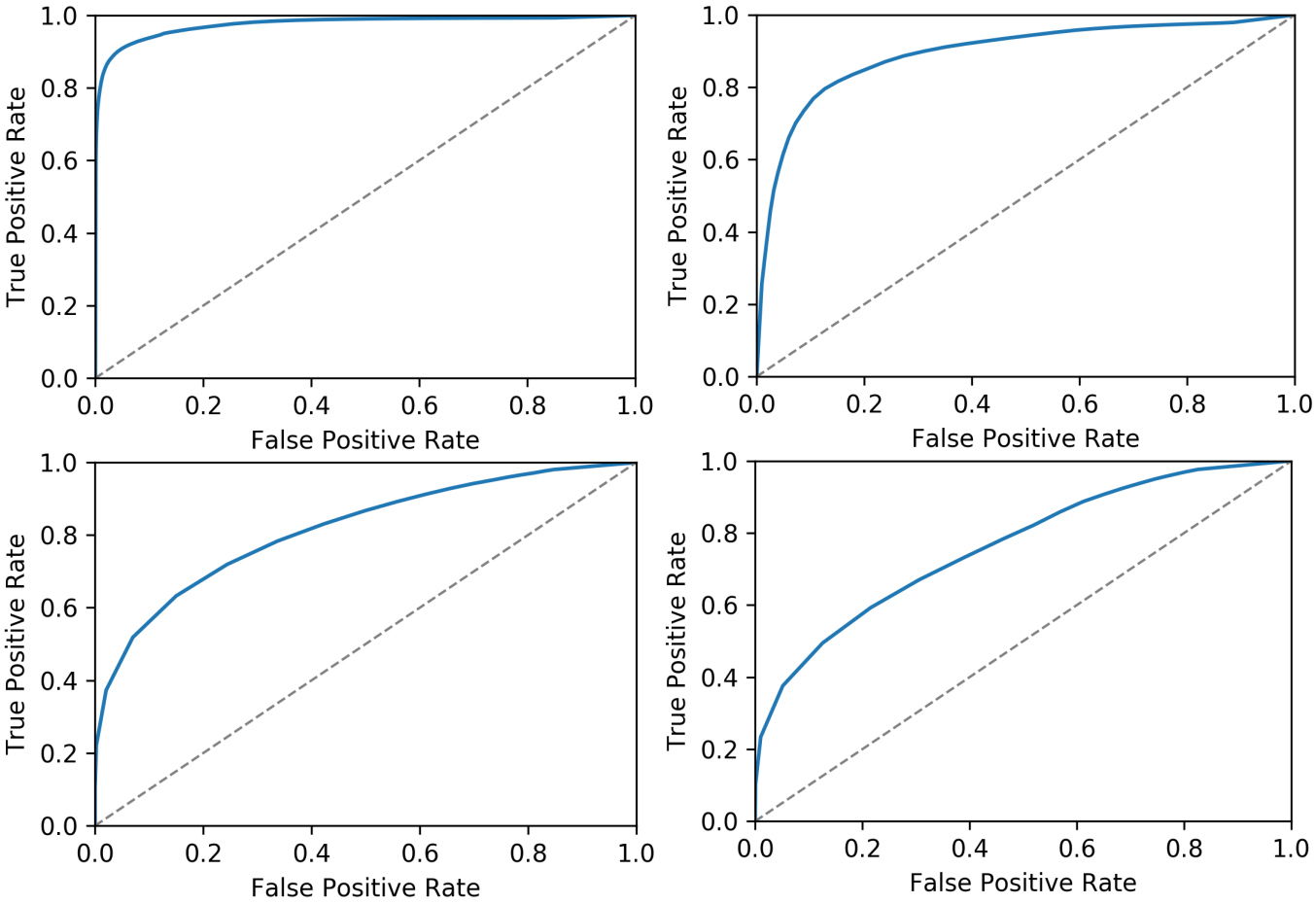

Fig. 2: ROC curves. Upper-left: proposed model on $S_1^{(test)}$. Upper-right: Proposed model on S_2. Lower-left: Baseline model on $S_1^{(test)}$. Lower-right: Baseline model on S_2.

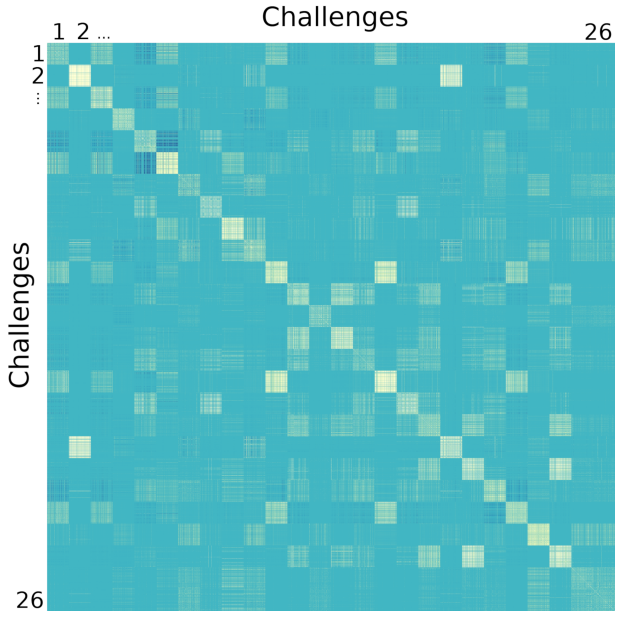

Fig. 3: Visualization of pairwise distances between challenges in S_2. Lighter colors represent closer distances.

REFERENCES

[1] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 18–33.

[2] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[3] C. Sung, C. Y. Higgins, B. Zhang, and Y. Choe, "Evaluating deep learning in chum prediction for everything-as-a-service in the cloud," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 3664–3669.

[4] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236085

[5] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 933–944. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180167

[6] 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*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[7] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[8] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C. Kim, "A deep tree-based model for software defect prediction," *CoRR*, vol. abs/1802.00921, 2018. [Online]. Available: http://arxiv.org/abs/1802.00921

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[10] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 354–365. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236026

[11] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 200–210. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196334

[12] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in neural information processing systems*, 1994, pp. 737–744.

[13] Y. Sun, Y. Chen, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, 2014, pp. 1988–1996.

[14] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *ICML Deep Learning Workshop*, vol. 2, 2015.

[15] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," *CoRR*, vol. abs/1503.03832, 2015. [Online]. Available: http://arxiv.org/abs/1503.03832

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[17] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[19] "Github," https://github.com/, accessed: 2018-05-30.

[20] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," 2016. [Online]. Available: http://arxiv.org/abs/1611.08307

[21] "Hackerrank," https://www.hackerrank.com/, accessed: 2018-05-30.

[22] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 476–480.

[23] "mypy," http://mypy-lang.org/, accessed: 2019-01-10.