ANALYSING FEATURES LEARNED USING UNSUPERVISED MODELS ON PROGRAM EMBEDDINGS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this paper, we propose a novel approach for analyzing and evaluating how a deep neural network is autonomously learning different features related to programs on different input representations. We trained a simple autoencoder having 5 hidden layers on a dataset containing Java programs, and we tested the ability of each of its neurons in detecting different program features using only unlabeled data for the training phase. For doing that, we designed two binary classification problems having different scopes: while the first one is based on the program cyclomatic complexity, the other one is defined starting from the identifiers chosen by the programmers, making it more related to the functionality (and thus, to some extent, to the semantic) of the program than to its structure. Using different program vector representations as input, we performed experiments considering the two problems, showing how some neurons can be effectively used as classifiers for programs on different binary tasks. We also discuss how the program representation chosen as input affects the classification performance, stating that new and customized program embeddings could be designed in order to obtain models able to solve different tasks guided by the proposed benchmarking approach.

1 INTRODUCTION

Mining information from source code has a crucial relevance in many fields of computer science. At a first glance, it can help software developers and software engineers in their tasks of error correction, code maintenance and code reuse. In addition to this, applications in other settings are also possible, such as in security analysis or for finding new and effective ways for teaching programming languages.

The aim of this work is to assess the ability of a generic machine learning model of building internal representations able to detect different and general program features, similarly to what other authors (Le et al., 2012) did for images.

We implemented a simple autoencoder and we trained it on different program embeddings. In spite to the fact that we are using program embeddings designed for specific purposes, the experiments we conducted show that some neurons are able to detect some program features, stating that our approach is promising and can be used as a baseline for further research.

2 BACKGROUND

In the literature, there exist several examples of tools or models focused on the extraction of information from source code or, more in general, from programs. This section provides a brief overview of the attempts in this direction, focusing on the perspective we are interested in, and gives an introduction to the works that most inspired our research statement.

A first interesting approach is the work of Allamanis et al. (2018) in which, by means of a graph based representation for programs derived from their abstract syntax trees, a deep learning model is trained for solving tasks such as predicting the name of a variable or detecting if a variable has been misused. A similar task is addressed in Alon et al. (2019) and Allamanis et al. (2016), where the authors test the performance in solving the method naming task by means of a source code

embedding (i.e. a vector representation for source code) and a convolutional neural network based on the transformer model (Vaswani et al., 2017), respectively.

The main motivation in studying approaches for predicting the name of a program component (e.g. a method or a variable) lies in the fact that, since in most cases that name is given by the programmer in order to give advice on the purpose of that component, being able to predict that name is somehow close to extracting some *semantic* information from a piece of code. A comparable purpose is targeted by Mou et al. (2016), where the authors produce a dataset of programs labelled according to their task and a deep learning-based embedding is used for the classification of the programs by functionality.

The idea of extracting information from the source code is also applied in other fields. For example, referring to cybersecurity, there exist many examples of the application of various machine learning techniques for detecting or classifying software vulnerabiliteis. Examples are proposed in Russell et al. (2018), where a supervised deep learning model is used for the classification of different kinds of software weaknesses, and in Yamaguchi et al. (2011), where a program embedding and common usage patterns are used for detecting code potentially vulnerable to a known weakness.

All the approaches mentioned before share the common objective of mining a specific information from programs. The aim of this work, instead, is to test if it is possible for a general machine learning model to build an internal representation for different program features after an unsupervised training process. This idea takes its inspiration from the work of Le et al. (2012), where the authors conducted experiments on images in order to build a feature representation from unlabelled data, showing how their model was able to effectively recognise, for example, faces and cats.

3 EXPERIMENTS

This section describes the experimental setting we adopted for testing the effectiveness of our approach. We first implemented two simple autoencoders (Goodfellow et al., 2016) for program vectors, and we trained them using two distinct source code embeddings as input. We then tested the ability of such autoencoders in learning different (and never seen before) program features, similarly to what Le et al. (2012) did for images.

3.1 NETWORK ARCHITECTURE

Since the aim of this work is to lay the foundations of a methodological approach for mining varied properties from a program, the study of the best network architecture is not in the scope of this paper. Therefore, without any hyper-parameters optimization nor any in-depth study on the network design, we implemented two simple dense autoencoders having two hidden layers in the encoder, two symmetrical hidden layers in the decoder and one code layer in the middle. We used the *ADADELTA* optimizer (Zeiler, 2012) and a Rectified Linear Unit (ReLU), defined, for all $x \in \mathbb{R}$ as $\max(0, x)$, as the activation function for the hidden layers. As a loss function, we used the Mean Squared Error:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x}_i - \boldsymbol{y}_i)^2 \tag{1}$$

Where n is the cardinality of the set of vectors, $\boldsymbol{x}_i$ is the i-th input vector and $\boldsymbol{y}_i$ is the prediction for the i-th vector.

The summary of our network architecture is shown in Figure 1. We wrote this implementation with the APIs provided by the Keras library (Chollet et al., 2015).

3.2 TRAINING

We trained our autoencoders on the Java dataset also used in Allamanis et al. (2016), which is a 2GB collection of popular gitHub[1] projects containing over 200000 methods. For each method in

[1] `https://github.com`

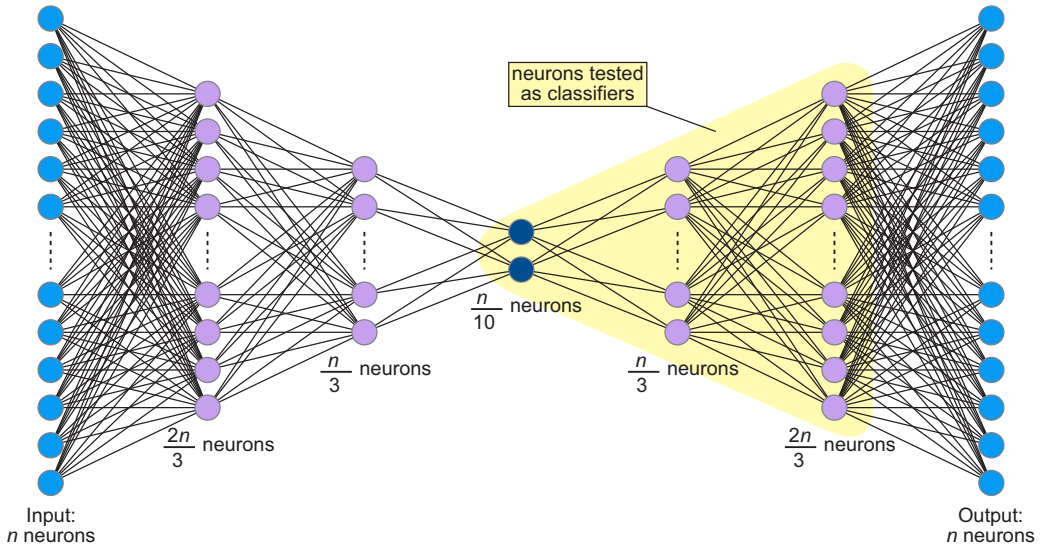

Figure 1: Architecture of the networks used for the experiments. The sizes of the layers differs in the models due to different embedding dimensions: we considered $n = 384$ for code2vec vectors and $n = 300$ for doc2vec vectors. Layers highlighted in yellow are those tested in the classification experiments.

the dataset, we built the corresponding vectors by applying two kinds of source code embedding techniques:

1. The source code embedding obtained with the code2vec model (Alon et al., 2019), which produces 384-dimensional code vectors combining the paths on the abstract syntax tree through the attention model (Vaswani et al., 2017).
2. A shallow 300-dimensional embedding obtained by applying the doc2vec model (Le & Mikolov, 2014) on the pure source code, without any preprocessing, using the Gensim (Řehůřek & Sojka, 2010) framework.

The reason for the choice of those embeddings as our inputs lies in the underlying construction ideas: the code2vec model is designed for predicting the name of a method, and thus for capturing some semantic information related to the method, while the other one is a rough application of a classical natural language processing algorithm (Mikolov et al., 2013) to the pure source code, which is not supposed, dealing with source code instead of natural language, to seize features or properties of any type.

3.3 Problems Definition

The focus of this work is to investigate the ability of a general deep neural network in building an internal representation able to detect different features after an unsupervised training phase. To this end, we defined two different binary classification problems having throughly different natures. The first one, which is designed using the program *control flow graph* (CFG) (Allen, 1970), addresses the structure of the source code, while the other one uses the identifiers chosen by the programmers in order to target a semantic task.

Structural problem We considered the *cyclomatic complexity* (McCabe, 1976) of a program, defined from its control flow graph $\mathcal{G}$ having n vertices, e edges and p connected components as:

$$V(\mathcal{G}) = e - n + p \tag{2}$$

Dealing with java methods, such metric can be easily calculated counting 1 point for the beginning of the method, 1 point for each conditional construct and for each case or default block in a switch-

case statement, 1 point for each iterative structure and 1 point for each Boolean condition. Starting from this software metric, we defined the problem as follows:

Problem 1 *Let M be a set of Java methods, and let $h_c : M \to \{0, 1\}$ be a classification rule for the methods. We define, for each $m \in M$:*

$$h_c(m) = \begin{cases} 0 & \text{if } V(\mathcal{G}_m) < 10 \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

Semantic problem For the definition of this problem, we considered the assumption (as in the works of Allamanis et al. (2016) and Alon et al. (2018)) that the name that a programmer gives to a method is a sort of summary of what that method does. This means that the name of a method shall providesome semantic information about the method itself. Starting from this premise, we defined this semantic problem using the *sub-tokens* in the method name (i.e. the words obtained by splitting the names considering the camel case and the snake case notations) and we locate a method in a class according to the presence or the absence of a given sub-token into its name. Formally, we can describe this binary classification problem as follows:

Problem 2 *Let N be a set of method names, and let T be a set of words. We write $s_1 \leq s_2$ if s_1 is a substring of s_2. Let $h_t : N \to \{0, 1\}$ be a classification rule for the methods. We define, for each $n \in N$ and for each $t \in T$:*

$$h_t(n) = \begin{cases} 1 & \text{if } t \leq n \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Random problem Finally, we defined a problem defined by random labels to be used as a benchmark. For this problem, we only considered a random split of the methods:

Problem 3 *Let L be a list of methods, and let $m_i \in L$ be the method having index i. Given a threshold n and a function $h_n : L \to \{0, 1\}$, we define, for each $m_i \in L$:*

$$h_n(m_i) = \begin{cases} 1 & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

3.4 CLASSIFICATION

In order to assess the ability of our network in autonomously learning program features, we tested the performance of the hidden neurons in classifying methods according to the problems described below.

Using the combinations between the program vectors described in Section 3.2 We considered the code layer and the two hidden layers in the decoder, as shown in Figure 1, and we tested the accuracy score of each neuron in classifying programs, considering different thresholds of the activation values.

The Algorithm 1 describes the procedure for finding the best neuron. We applied such routine to both our models (i.e. the autoencoders trained on the code2vec and doc2vec models, respectively) using, for each binary classification problem, a balanced random sample of the dataset containing 1500 positive examples and 1500 negative examples, always obtaining an accuracy score higher than 60%, as reported in Table 1.

In order to verify the significance of the result, namely the fact that such accuracy scores are reasonably higher than those of a "lucky" random classification, we compared them with an actual random one: we first produced a balanced 300-dimensional binary vector $\boldsymbol{x}$ and then, considering $\boldsymbol{x}$ as the vector representing the true labels, we simulated 5000 predictions using 5000 randomly generated binary vectors. Such experiment produced a best accuracy score of 53%, stating that our best neuron is a significantly better classifier than a pure random one.

Algorithm 1: Algorithm for finding the best neuron in classifying programs

```
bestAccuracy = -1 // Accuracy of the best neuron
vectors := list of program vectors
for each neuron do
    activations := list of activation values for that neuron
    minVal = 0
    maxVal = max(activations)
    step = (maxVal - minVal)/10
    for i = 0 to 10 do
        threshold[i] = i*step
    end
    best = -1 // Best accuracy for this neuron
    for each threshold do
        predLabels := predicted class labels
        for i = 0 to len(activations) do
            if activations[i] < threshold then
                predLabels[i] = 0
            else
                predLabels[i] = 1
            end
        end
    end
    if accuracy(predLabels) > best then
        best = accuracy(predLabels) // update best accuracy for this neuron
    end
    if best > bestAccuracy then
        bestAccuracy = best // update global best accuracy
    end
end
return neuron having best accuracy
```

Table 1: Accuracy obtained by the best neuron for the considered problem instances.

Problem	**code2vec**	**doc2vec**
Random	58%	56%
Complexity	63%	65%
Semantic - find	67%	68%
Semantic - create	64%	57%
Semantic - find OR create	65%	60%

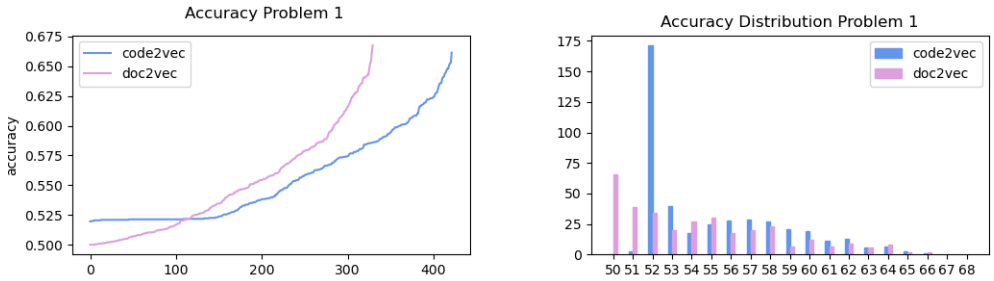

Figure 2: Results for the structural problem. The plot to the left reports the accuracy for each neuron, while the plot on the right show the accuracy distribution with respect to the number of neurons that achieve a certain accuracy level.

4 Results discussion

Since our goal was to start playing with analysis methods derived from works on images (e.g. Le et al. (2012)), and adapting it to the specific setting of unsupervised models on program embeddings, we are looking for evidences that those methods, even in the simple models we used, are able to discover neurons, possibly internal ones, which perform significantly better than others, on tasks not directly related to the unsupervised learning goals of the model.

As introduced in the previous section, we trained a simple autoencoder on vectors from two different embeddings: code2vec and doc2vec, and later we fed as input for inference to each of the trained autoencoders, two possible sets of vectors: one defining a supervised task on a code structural problem, and a second one for a code semantics problem. We are not looking for a good performance of our models on these tasks, instead we are looking for (possibly internal) neurons whose activation values on the input vectors allow to obtain a classification accuracy significantly better (above 60%) than a random labeling. Interestingly, we consistently found them in the 4 experimental settings (2 unsupervised learners, each measured against 2 different supervised tasks).

Finally, we checked the "best neuron" behaviour of the trained models when input vectors were randomly labeled. Surprisingly, no neuron has been able to classify such artificial data sets with accuracy above our threshold of 60%.

More in details, for the Structural problem, the model based on code2vec performed worse than the doc2vec model. The accuracy on correct labels was clearly above 60%, but the accuracy on random labels stayed slightly below our threshold. On the other hand, the model based on doc2vec had a similar performance on the correct labels, but a better separation of results on random labels: in this case the best accuracy of its neurons stayed more clearly below our threshold, typically around 55%. See Figure 2.

Considering the Semantic problem, we tested our approach on three problems in the category. We checked our 2 trained systems on three data sets which classified, as positive instances, methods (their embeddings) whose identifiers were related to some chosen operation. Viz, the three data sets have been those for: methods performing some *find* operation, methods performing some *create* operation, and finally the data set where positive instances are the union of those from the other two data sets.

The goal was to check the performance on two expectedly different semantic contexts (and the source codes implementing the *find* and *create* are typically different), and finally to check whether any neuron in the trained models was able to in some way classify instances from the union problem, thus recognizing the more difficult concept which is the "OR" disjunction of the first two.

The results on these three different examples of the Semantic problem showed what follows. Both learning systems had neurons able to classify reasonably well the instances of the three data sets, but with different quality on the first two, and worse in the third "union" data set. The performance on the three data sets can be seen in Figure 3.

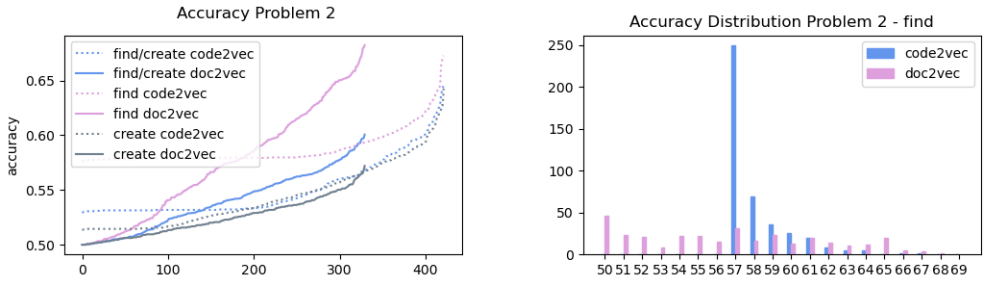

Figure 3: Results for the semantic problem, considering different identifiers. The plot to the left reports the accuracy for each neuron, while the plot on the right show the accuracy distribution with respect to the number of neurons that achieve a certain accuracy level.

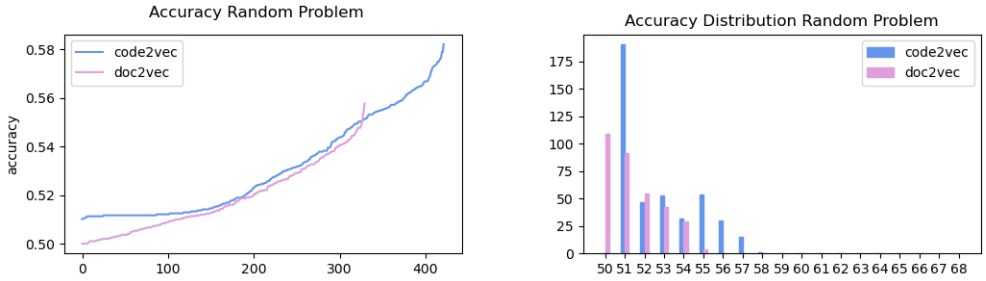

Figure 4: Results for the random problem. The plot to the left reports the accuracy for each neuron, while the plot on the right show the accuracy distribution with respect to the number of neurons that achieve a certain accuracy level.

Considering the 2 different high level tasks and their union, *find* and *create*, the best performance has been with *find*, compared to *create*, but more interestingly, the worse behavior was for the task requiring to recognize their union.

Finally, we find useful to visualize the overall behavior of our systems on the random problem in Figure 4.

5 Conclusion

We defined and tested methods to analyse the behavior of trained unsupervised models for programs code, when their neurons are tested as classifiers for labeled problems, in analogy with what has been done in a similar setting for images.

Results have been interesting, because we discovered neurons which could do significantly better than a random classifier on problems related to the structure and to the semantic of source code instances. The interest comes from the fact that those "best neurons" are in a network trained and tested on input vectors which actually are themselves generated through a preceding embedding process for Java source code methods. Therefore, even if their performance on structural and semantic classification problems is very limited (with accuracy around 66%), their activation values seem to somehow distinguish embedding vectors coming from source code having different structural, or semantic, properties.

This approach could be further studied, because in some cases we need a good explanation of why such (approximate) classification is emerging, even if performed on the embedding and not on the source code (or its abstract syntax tree). For instance, for our experiments we used embeddings defined in literature, which could perhaps explain some of our results, because they have been built with goals good for our simple tasks (consider the identifiers for code2vec and our Semantic problem). But this is not the case for all of our experiments.

Therefore, we could first consider different embeddings, so to favour the passing of useful information from source code through to the learning neurons, and also we will study more sophisticated learning models, which will be actually designed for performance, so to check in that more realistic setting whether "good neurons" emerge.

REFERENCES

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, pp. 2091–2100, 2016.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of 6th International Conference on Learning Representations, ICLR 2018*, 2018.

Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pp. 404–419, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

François Chollet et al. Keras. `https://keras.io`, 2015.

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, pp. 1188–1196, 2014.

Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of the 29th International Conference on Machine Learning, ICML*, 2012.

Thomas J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26, Proceedings of NIPS 2013*, 2013.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 1287–1293, 2016.

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*, pp. 45–50, 2010.

Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA*, pp. 757–762, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Proceedings of NIPS*, pp. 5998–6008, 2017.

Fabian Yamaguchi, Felix "FX" Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of 5th USENIX Workshop on Offensive Technologies WOOT 2011*, pp. 118–127, 2011.

Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.