Learning student program embeddings using abstract execution traces

Guillaume Cleuziou
University of Orléans, INSA Centre Val de Loire,
LIFO EA 4022
Orléans, France
guillaume.cleuziou@univ-orleans.fr

Frédéric Flouvat
University of New Caledonia, ISEA EA 7484
Nouméa, New Caledonia
frederic.flouvat@unc.nc

ABSTRACT

Improving the pedagogical effectiveness of programming training platforms is a hot topic that requires the construction of fine and exploitable representations of learners' programs. This article presents a new approach for learning program embeddings. Starting from the hypothesis that the function of a program, but also its "style", can be captured by analyzing its execution traces, the *code*2*aes*2*vec* method proceeds in two steps. A first step generates abstract execution sequences (AES) from both predefined test cases and abstract syntax trees (AST) of the submitted programs. The *doc*2*vec* method is then used to learn condensed vector representations (embeddings) of the programs from these AESs. Experiments performed on real data sets shows that the embeddings generated by *code*2*aes*2*vec* efficiently capture both the semantics and the style of the programs. Finally, we show the relevance of the program embeddings thus generated on the task of automatic feedback propagation as a proof of concept.

Keywords

Representation Learning, Program Embeddings, Neural Networks, Educational Data Mining, Computer Science Education, doc2vec.

1. INTRODUCTION

Increasingly, programming is being learned through the use of online training platforms. Typically, learners submit their code(s) and the platform returns any syntax errors or functional errors (typically based on test cases defined by the teacher). The exploitation of these data opens up new perspectives to monitor and help beginners in learning programming. They can be used, for example, to identify students who are dropping out, to target bad practices or to propagate teacher feedbacks. These functionalities would allow the learner to be more autonomous during his learning, and the teacher to be more reactive and efficient in his interventions. However, this exploitation requires a detailed analysis of the submitted programs. These training platforms must go beyond a simple syntactic analysis of the script, and allow the associated semantics to be considered. For this purpose, learning program embedding has recently emerged as a promising area of research [15, 13, 17, 7, 2, 3]. A natural way to generate such vectorial and condensed representations is to consider a computer program as a text and to exploit methodologies inspired by Text Mining.

Text mining has attracted a lot of interest in recent years. The representation of texts as vectors of real numbers, also called "embedding", has been at the heart of many recent works. These representations make it possible to project (or 'embed') a whole vocabulary into a low-dimensional space. Moreover, such a representation of words allows to exploit a wide variety of numerical processing methods (neural networks, SVM, clustering, etc.). At that stage, one of the challenges is to capture in these representations the underlying semantic relationships (e.g. similarities, analogies). The work of [11] based on the use of neural networks has been a precursor in this area. Their *word*2*vec* method is one of the most referenced in the field. Its principle is based on the relation between a word and its context (words appearing before and after). To do this, they propose some simple and efficient architectures to learn word embeddings from a corpus of texts. For example, the *CBOW* (Continuous Bag-Of-Words) architecture trains a neural network to predict each word in a text given its context. Their results show the ability of this approach to extract complex semantic relations (analogies) from simple operations on $v()$ projections, s.t. $v(\text{"}king\text{"}) - v(\text{"}man\text{"}) + v(\text{"}woman\text{"}) \approx v(\text{"}queen\text{"})$ or $v(\text{"}Paris\text{"}) - v(\text{"}France\text{"}) + v(\text{"}Italy\text{"}) \approx v(\text{"}Rome\text{"})$.

The transposition of these approaches to computer programs is not straightforward. The code has certain specificities that need to be integrated to have such rich representations [1]. Unlike texts, codes are runnable, and small modifications can have significant impacts on their executions. A program can also call other programs that can themselves call other programs. The context in which an instruction is used is also particularly important in deducing its role. Finally, unlike texts, program syntax trees are usually deeper and composed of repeating substructures (loops). Existing approaches for building program embeddings only partially integrate these specificities. They independently exploit the instructions [3], the inputs/outputs [15], part of the execution traces [17] or the abstract syntax tree (AST) [2]. They

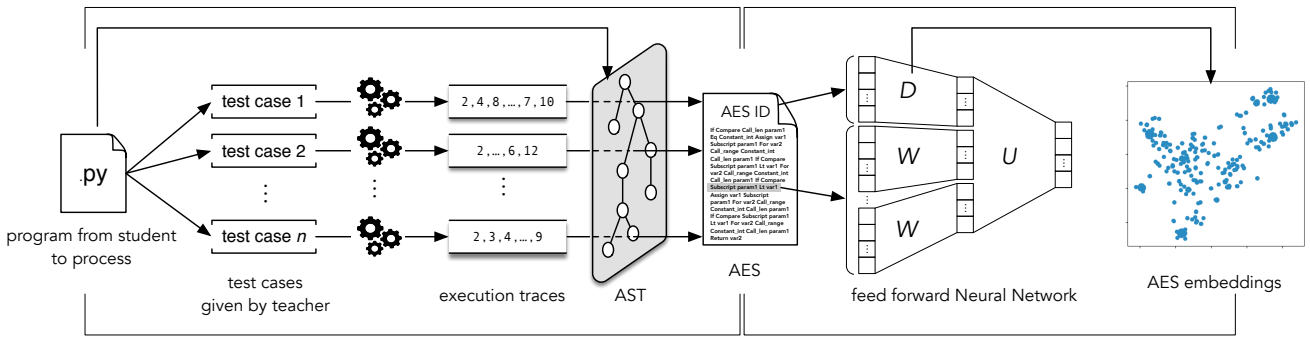

Figure 1: General scheme of the *code2aes2vec* method for learning program embeddings.

focus more on the function of the program (what it does) than on its style (how it does). Moreover, most of these approaches are supervised and build embeddings for a specific task (e.g. predict errors, predict functionality, etc.).

In view of these limitations, we propose the *code2aes2vec* method, exploiting instructions, code structure and execution traces of programs, in order to build finer program embeddings. Figure 1 schematizes the overall approach *code2aes2vec* we propose. The first step of this method consists in generating Abstract Execution Sequences (AES) from traces obtained on test cases executions and program ASTs. The second step uses the *doc2vec*[1] neural network [8] to learn program embeddings from AESs. Contrary to existing approaches, we therefore propose a generic and unsupervised method that learns program embeddings by using functional, stylistic and execution elements. This aspect is crucial in our application to be able to differentiate programs answering to the same exercise (i.e. implementing the same functionality) but in different ways (in terms of strategy or efficiency). Our approach is validated on two real data sets, composed of several thousands of Python programs from educational platforms. On these datasets, we show that embeddings generated with *code2aes2vec* allow to efficiently detect the function and the style of a program. In addition, we present a proof of concept of the use of such embeddings to propagate teacher feedbacks.

To summarize, the main contributions of this paper are :

1. the definition of a new (intermediate) program representation, called Abstract Execution Sequences (AES), allowing to capture more semantics,
2. exploiting these (intermediate) representations with *doc2vec* to build program embeddings in an unsupervised way,
3. the diffusion to the community of two enhanced datasets from educational platforms in computer science,
4. a proof of concept on the use of such program embeddings for feedback propagation for educational purposes.

[1]a method derived from *word2vec* that allows to learn a document embedding from its words.

The next section details existing works in the field and the originality of our approach in relation to it. Section 3 presents our two-steps method: the construction of abstract execution sequences (AES) and the learning of embeddings from them. Section 4 is devoted to the qualitative and quantitative evaluations of the learned representations before drawing up the many perspectives of this work (section 5).

2. RELATED WORKS

Learning representations from programs is at the heart of many recent works. They aim to embed this data in a semantic space from which further analysis can be conducted, because the generated representations (vectors of real values) are directly exploitable by a large part of the learning algorithms. To do this, recent work relies heavily on methods developed to build word embeddings in texts, while trying to integrate the specificities of the code.

These program embeddings are then used for prediction or analysis tasks related to the software development (debugging, API discovery, etc.) and teaching (learning programming). Two types of embeddings are more particularly studied: embeddings of the elements composing a program (words, tokens, instructions, or function calls) [12, 6, 7] and embeddings of the programs themselves [15, 17, 3, 4].

Nguyen et al. [12] study API call sequences to derive an API embedding that is independent of the programming language (thus allowing translation from one language to another). This embedding is learned by *word2vec* [11] from call sequences from several million methods. This method allows to build a word embedding from words appearing nearby in each text (i.e. its context). Two datasets derived from the Java JDK and from more than 7000 recognized C# projects from GitHub (more than 10 stars) are used for learning.

De Freez et al. [6] have a close objective, namely to build an embedding of functions used in a code in order to find

synonymous functions. However, the function calls made in each program are extracted in the form of a graph depending on the control structures. Random walks are then performed in this graph for each function, and the extracted paths are used as input to *word2vec* [11] to derive an embedding of functions. An extract of two million lines of the Linux kernel is used to learn the embedding.

In [7], the authors propose a relatively similar approach but replace function calls with abstract instructions. Each instruction sequence represents a path in the code depending on conditional instructions. In this way, it is similar to a trace even if the code is never executed. All possible paths are extracted but loop repetitions are ignored and only the most frequent instructions are considered (a threshold of 1000 occurrences is used in the experiments). Moreover, only certain constant values are considered to limit the size of the vocabulary. An embedding of these program instructions is then learned by the GloVe method [14] based on the co-occurrence matrix of the words. The authors also use a corpus of 311,670 procedures (in C language) from the Linux kernel, and evaluate it on a data set of 19,000 pre-identified analogies.

In many applications, it is not just a matter of considering the program's components but the program as a whole. Recent work has therefore focused on the construction of program embeddings.

Following a 'teaching' motivation, Piech et al. [15] construct student program embeddings and use them to automatically propagate teacher feedbacks (using the *k-means* algorithm). The embedding space is built from a neural network trained to predict the output of a program using its input. It thus captures the *functional* aspect of the code. The authors also try to capture the *style* of programs, using a recursive neural network based on the program's AST. Contrary to other approaches, the generated embeddings are matrices, not vectors, thus limiting their exploitation as inputs for next data analyses. They also don't consider learner-defined variables. The obtained representations actually capture quite well the code *function*, but fail in capturing the *style* of codes. The analyzed programs (from the Hour of Code site and a course at Stanford) are written in a language similar to Scratch and allow operations in a labyrinthine world.

In [17], the authors highlight the limitations of syntax-based approaches to capture the semantics of a program. Instead, they propose to consider the trace resulting from the code execution, and more precisely the values of the most frequent variables. Different representations are proposed and used to train a recurrent neural network whose objective is to predict errors made by students in a programming course. The embeddings of the programs corresponds to one of the neural network's layers. The authors put forward one representation more particularly, considering the trace of each variable independently and integrating the dependencies between variables in the structure of the neural network. However, the obtained embeddings are specific to one task, and this method requires to redefine the neural network architecture, with re-training, for each exercise.

Finally, Alon et al. [2] propose a neural network to predict the name of a method (i.e. the functionality) from its code. To do this, the program is first decomposed into a collection of paths (from one leaf to another) in the AST. Only the most frequent paths in the dataset are used as features (size constraints are also integrated). Then, the network learns which one is important for predicting the method name using the attention principle. The parameters of the trained neural network correspond partly to the final embeddings and partly to the weights supposed to quantify the importance of each (feature) path for the prediction task (attention principle). Training is performed on a corpus of more than 13 million Java programs from GitHub's 10,072 most popular projects. As mentioned by the authors, this approach requires a large number of input programs. Furthermore, it is not possible to predict the function (and embedding) of a program whose paths do not appear in the training set. The embeddings produced capture information and semantic relationships about the function of the code, but ignore style variants. Thus, two programs with the same function will be similar, regardless of how they have been coded. The quality of the analyzed code also has an impact on learning. The names given to the variables are particularly important for prediction.

3. THE *code2aes2vec* METHOD

Two main strategies emerge for learning program embeddings : by observing the results of program execution [15, 17] or by analyzing the script [3] and/or its AST [2]. Our approach is at the intersection of these two strategies and thus aims to take advantage of the functional and syntactic descriptions of the programs to induce relevant embeddings. We thus propose the *code2aes2vec* method which proceeds in two steps :

1. the *code2aes* step represents a program as an Abstract Execution Sequence (AES), corresponding to the AST paths used by the program during its execution on predefined test cases;

2. the *aes2vec* step uses a neural network to construct the embedding of the programs based on their AES (using the *doc2vec* approach [8]).

3.1 *code2aes*: construction of Abstract Execution Sequences (AES)

Translating a program into an AES requires providing, in addition to the program itself, a collection of test cases on which the program will be run in order to exploit its traces. In our educational context, the preparation of such a collection of test cases is not an additional effort since test cases are generally integrated into training platforms to evaluate submitted contributions. Moreover, this approach offers teachers the possibility of introducing verification choices and thus to drive the interpretation of his/her learners' programs according to his/her own pedagogical choices. For example, let's consider an exercise whose objective is to find a value in a table/list. A teacher wishing to emphasize algorithmic efficiency may choose to integrate a few unit tests for which the desired value appears early in the table. In such case, an efficient program stops the loop as soon as the desired value appears. These test cases will thus make it

possible to distinguish two (valid) programs based on their execution trace.

In practice the number of test cases provided by the teacher is quite small. We generally observe that less than ten test cases are enough to evaluate whether a program is correct or not.

Figure 2 illustrates the process of translating a program into an AES. This example considers as input the code submitted by a learner in response to the exercise "*write a Python function that returns the minimum value in an input list*". First, the AST is constructed. It describes the syntactic structure of the program in terms of control structures (`if-else`, `for`, `while`), function calls (`call`), assignments (`assign`), etc. Second, the code is executed on an example (here the input `[12, 1, 25]`) and its execution trace is kept, indicating the program lines successively executed. Finally, the AES is constructed by mapping these two levels of information: syntactic and functional. The sequence resulting from the trace is translated into a sequence of "words" extracted from the nodes of the AST.

Three levels of translation (or abstraction) are proposed according to the depth considered in the AST:

- AES level 0 : each program line is represented by a single word corresponding to the head symbol of the associated sub-tree in the AST (in red in Figure 2),
- AES level 1 : each program line is represented by one or more words corresponding to the head symbols of the associated sub-tree and its main sub-trees (in red and blue in Figure 2),
- AES level 2 : each program line is translated in a sequence of words corresponding to all the nodes appearing in the associated sub-tree (in red, blue and black in Figure 2).

For the last two levels, the names of the variables and parameters, as well as the values of the constants, have been normalized so as not to artificially extend the considered "vocabulary". Thus, the variable `res` is renamed `var1` and the variable `i` is renamed `var2`.

Note that each execution of a program on one test case generates a partial AES. A program will finally be represented by the concatenation of the partial AES obtained on each test case. An AES can thus be considered as a representative text of the program. Each partial AES corresponds to a sentence of this text.

3.2 *aes2vec* : learning program embeddings from AES

The *word2vec* method [11] is based on the distributional hypothesis of words in natural language [16] : a word can be inferred from its context. For example, the CBOW (Continuous Bag Of Words) version of *word2vec* allows to train an feed-forward Neural Network to predict a (central) word from its context. In *word2vec* a *context* is defined by the preceding and following words in the text. The structure of the neural network is reduced to a single hidden layer (encoding) ; the matrix W of the weights connecting the input layer to the hidden layer contains the word embeddings.

word2vec has already been used to learn token embeddings from a computer program [6, 3]. However the distributional hypothesis seems less satisfied on the tokens from a program than on the natural language, in particular because of the very limited size of the vocabulary and especially a little constrained compositionality (almost all combinations are observed).

[8] have proposed a variant of *word2vec*, aiming to learn simultaneously the embeddings of the words and the documents from which they are extracted. The *doc2vec* method is still based on the distributional hypothesis allowing to predict a word knowing its context, but this time the context integrates (in addition to the preceding and following words) the identifier of the document from which the word sequence comes from. In doing so, the authors introduce the idea that there are document specific variations in the natural/universal distribution of words in the language.

We exploit precisely this hypothesis of *document-based distributional variations* for the processing of AES built from the programs. We consider that each program, during its execution, generates different sequences of tokens (AES). We then use the DM (Distibutive Memory) version of the *doc2vec* algorithm to train a feed-forward Neural Network (with one hidden layer) to maximize the following log probability :

$$\mathcal{L} = \sum_{s=1}^{S} \sum_{i=k}^{T_s - k} \log p(w_i^s | w_{i-k}^s, \ldots, w_{i+k}^s, d_s) \qquad (1)$$

with S the total number of documents (or AESs), T_s the total number of words (or tokens) in document s, d_s the s^{th} document, w_i^s the i^{th} word in document d_s and k the size of the context on either side of the target word.

Figure 3 presents the architecture of the neural network used for *doc2vec* as we use it for learning program embeddings via their AES. The forward pass consists in first calculating the values of the hidden layer by aggregating the encodings of each word of the context and of the document : $h(w_{i-k}^s, \ldots, w_{i+k}^s; W, D)$ where $h()$ denotes an aggregation function to be defined (typically a sum, average or concatenation), W and D denoting the word embedding matrix (weight matrix between word inputs and hidden layer) and the document embedding matrix (weight matrix between document input and hidden layer) respectively. The output of the neural network can be interpreted as a probability distribution on the words of the vocabulary by applying an activation function (*softmax*) on the output $y_{w_i^s} = b + Uh(w_{i-k}^s, \ldots, w_{i+k}^s; W, D)$ where b is a bias term and U the weight matrix between hidden and output layer:

$$p(w_i^s | w_{i-k}^s, \ldots, w_{i+k}^s, d_s) = \frac{e^{y_{w_i^s}}}{\sum_j e^{y_j}} \qquad (2)$$

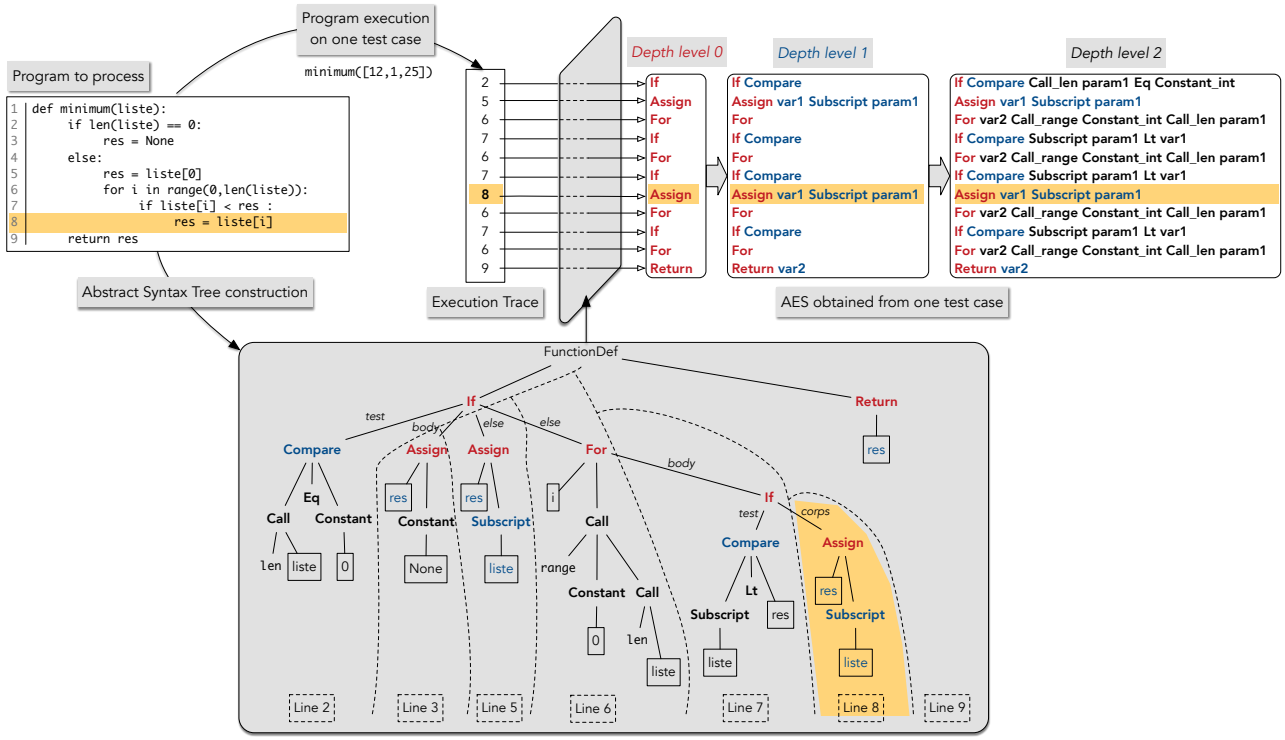

Figure 2: Construction of an AES from a Python program and a given test case.

Finally, the network weights are updated by stochastic gradient descent on the error, defined by the difference between the obtained output and the one-hot vector encoding the target word.

We consider programs as textual documents whose word sequence is given by an AES obtained by the previous step (*code2aes*). Indeed, as for text, the choice and the order of the "words" in an AES capture the semantic of the program, i.e. what the program does (its function) and how it operates (its style).

In this learning model, each AES vector (in D) is used only for predictions of the tokens from this AES, while token vectors (in W) are common to all AESs. The size of the vectors (for AES and tokens) is fixed and actually corresponds to the size of the desired representation space (embeddings).

Once the model is trained, the D matrix contains the embeddings of the programs. The positioning of a new program in this embedding space consists in inferring[2] a new column vector in D using the tokens from the new AES. The other parameters of the model remaining fixed (W as well as the *softmax* parameters).

Finally, let us mention that the choice of the aggregation strategy used in the hidden layer can be decisive. Indeed, a *sum* or *average* will consider each context as a bag-of-words (without taking into account the order), whereas a *concatenation* strategy offers the opportunity to exploit the order of words within the context. If the sequentiality (inside the context) is not a determining factor in the construction of embeddings for natural language, we will confirm in future experiments that the order of tokens is of high importance for learning program embeddings using AESs.

[2]Inference is made by purshasing the learning on the Neural Network.

4. EVALUATION OF THE APPROACH

4.1 Dataset presentation

Educational data are complex since programs may contain errors, be small in size, may not fully meet the intended functions and may be relatively redundant. These data have very different characteristics from the datasets used in software development. For our experiments, we thus built and use several real educational datasets (see Table 1). They consist of Python programs submitted by students on two training platforms in introductory programming courses. In addition to our (documented) *code2aes2vec* code for learning program embeddings, we also make available[3] these three "corpora" of Python programs, the associated test cases, and the AESs built on each program. All of the results presented in the rest of this section can thus be fully and easily reproduced.

The **NewCaledonia-5690** dataset (or NC-5690) includes the programs created in 2020 by a group of 60 students from the University of New-Caledonia, on a programming training platform[4]. The **NewCaledonia-1014** dataset (or NC-1014)

[3]*https://github.com/GCleuziou/code2aes2vec.git*

[4]Platform developed and made available by the CS department of the Orléans University Institute of Technology.

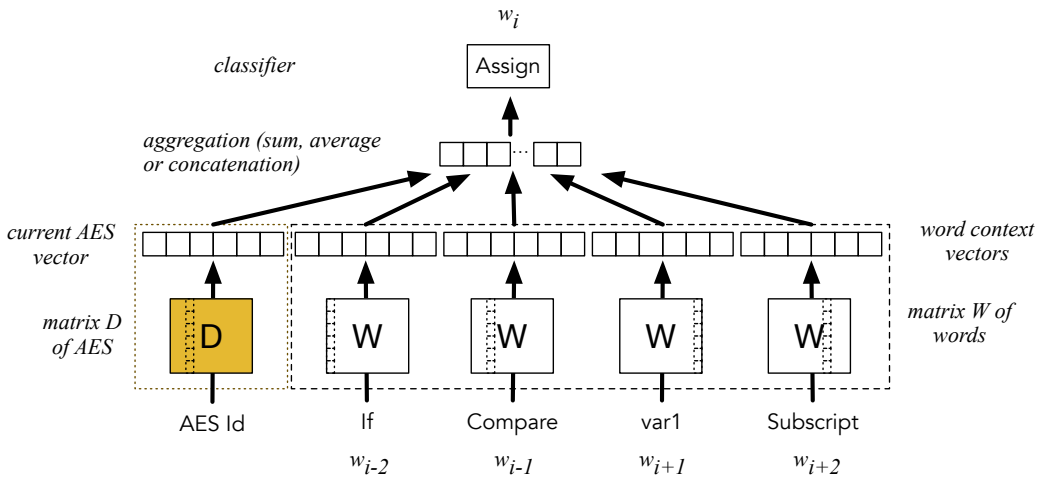

Figure 3: The neural network $aes2vec$ used to predict a word w_i from its original AES identifier and its context (here, two previous and two following words).

Table 1: Characteristics of the three Python datasets collected on programming training platforms. The 'Nb. of words' reported is the total size of the AES corpus; the number in parentheses indicates the size of the 'vocabulary'.

Datasets	NC-1014	NC-5690	Dublin-42487
Nb. programs	1,014	5,690	42,487
Avg nb. test cases per program	13.1	10.4	3.7
Nb. correct programs	189	1,304	19,961
Nb. exercises	8	66	65
Nb. 'words' AES-0	113,223 (20)	761,726 (44)	7,4 M (38)
Nb. 'words' AES-1	226,682 (42)	1,7 M (57)	15,2 M (83)
Nb. 'words' AES-2	690,019 (71)	3,9 M (113)	40,4 M (209)

Table 2: Exercises in the NewCaledonia-1014 dataset.

Exercice	Statement	#test cases
swapping	swap items in a list	4
minimum	look for the minimum in a list	8
compareStrings	compare two strings	11
fourMore100	return the first four values greater than 100 from an input list	7
indexOccurrence	return the index of the first occurrence of an item in a list	7
compareDates	compare two dates from their day, month and year	30
polynomial	return the roots of a polynomial of degree 2	6
dayNight	display information about the period of a day given a time	32

includes a sub-part of **NewCaledonia-5690** composed of contributions associated with 8 exercises selected for their algorithmic diversity (see Table 2) and their balanced volumetry (100 to 150 programs per exercise). We will use it as a 'toy' dataset facilitating qualitative analyzes. The **Dublin-42487** dataset includes student programs from the University of Dublin, carried out between 2016 and 2019. Although the original corpus [3] contains nearly 600,000 programs (Python and Bash), we propose here a subset enriched semi-automatically with test cases (not provided initially).

4.2 Embedding analysis

In the following experiments, each dataset has been divided into three sub-parts: training (90%), validation (5%) and test (5%); the validation set being used to select the best model among those learned during the different iterations ($aes2vec$). Unless otherwise stated, the $aes2vec$ algorithm has been set up to learn embeddings of dimension 100, the size of the context is set to 2, concatenation is used as aggregation stage and the training is performed over 500 iterations.

In a first step, we evaluate our approach in a qualitative way on the dataset **NewCaledonia-1014** constituted for this purpose. Figure 4 (left) shows a visualization of the 912 programs of the training set, obtained by a non-linear projection using the t-SNE dimension reduction algorithm [9]. It can be seen that, although embeddings are learned in an unsupervised manner, the $code2aes2vec$ method learns, from a relatively limited number of training data, a representation space in which the areas identify distinct program functionalities. Thus, the program vectors are organized quite naturally into 8 clusters that are highly correlated with the original 8 exercises. Moreover, the topological organization of the clusters matches well with the algorithm inherent to the programs. Exercises 'swapping' and 'minimum' are close in the embedding space and correspond to the only two exercises that iterate over all values of an input list. Exercises 'compareStrings', 'fourMore100' and 'indexOccurrence', in

the upper part of the space, require to partially iterate over a list. Finally, the last three exercises do not require loops and rely only on the use of conditional instructions. The exercise 'dayNight' is distinguished by the expected use of a display function (`print`) while all the other exercises return a result (`return`). This feature may explain the 'isolation' of the programs from this exercise from other programs.

Stylistic differentiation of programs is difficult to assess quantitatively. This task would require either objective criteria that can be extracted automatically or expensive expert labeling. Given the absence of such stylistic knowledge in datasets, we choose to illustrate on an example how the *code*2*aes*2*vec* method distinguishes different styles in the writing of a same function. Figure 4 (right) presents in detail the embedding space learned for the exercise 'minimum'. It's interesting to notice that program styles are clearly distinguished, notably the two ways to program a Python `for` loop (using indexes vs. elements directly). In a very detailed way, programs are also grouped according to whether their loop starts with the first element of the list (`range(0,...)`) or with the second one (`range(1,...)`), after initialization of the minimum to the first element in any cases.

Word order is important in our *aes*2*vec* method. For example, our approach distinguishes programs having a similar boolean condition but expressed in a different order (`if liste[i]<res:` vs. `if res>liste[i]:`). This distinction may seem artificial since these two expressions are strictly equivalent from the evaluation point of view. However, the first syntax appears more 'natural' than the second. It would be easy to get rid of this phenomenon by normalizing the expressions at the *code*2*aes* step; this option can be left at teacher's discretion.

Finally, we draw the reader's attention on some valid but atypical programs. In particular a program using the native Python function `min`, or the one using the `sort` function. Their separation from the rest of the programs is crucial since it offers a way to detect programs (a priori valid) that a teacher would like to reject or at least moderate considering that they deviate from his/her pedagogical objective. More generally, this analysis seems to confirm that the embedding spaces learned by the *code*2*aes*2*vec* method correctly captures not only the function of the programs but also their style. Their intrinsic quality paves the way for many practical uses that could significantly improve the efficiency of learning platforms (detection of atypical solutions, automation/propagation of feedbacks, student 'trajectory' analysis, study of error typologies, etc.).

In a second step, we evaluate the *code*2*aes*2*vec* approach from a quantitative point of view on the three datasets (Table 3). We consider an usual task for program embedding evaluation, namely the prediction of its function (i.e. exercise identification). For each considered configuration (AES level), the training data are used first to learn (without supervision) a representation space. Then, these embeddings are used to learn (with supervision) a SVM classifier (with polynomial kernel) [5]. Finally, the embeddings are (indirectly) assessed according to their ability to predict the function of the code (i.e. the exercise) for test data.

As baselines, *random classifier* informs about the difficulty of this task *a priori*, while *doc*2*vec* corresponds to the (naive) use of the algorithm *doc*2*vec* [8] to learn embeddings from the codes directly (without any intermediate representation). We also report the results obtained by the (supervised) *code*2*vec* approach[5] [2] executed with default parameters.

	NC-1014	NC-5690	Dublin-42487
random classifier	*0.125*	*0.015*	*0.009*
*code*2*vec* [2]	0.230	0.098	0.037
*doc*2*vec* [8]+SVM	0.412	0.495	0.380
*code*2*aes*2*vec*+SVM			
(AES-0)	0.882	0.460	0.391
(AES-1)	**1.0**	0.698	0.544
(AES-2)	**1.0**	**0.832**	**0.651**

Table 3: Quantitative and comparative evaluation of the produced embeddings, on the task of retrieving the function of a program (accuracy).

It can be seen that the *code*2*vec* model recently proposed by [2] cannot be trained satisfactorily on any of the three datasets. This is due to the numerous parameters to learn and the large number of examples this method requires. In the largest dataset (Dublin-42487), *code*2*vec* "only" has 42,487 programs as inputs. To the opposite, our *code*2*aes*2*vec* method as several million entries thanks to our AES intermediate representation.

The comparative results obtained with three different levels of AES (denoted by AES-0, AES-1 and AES-2 in Table 3) confirm that the quality of the embeddings is improved when the level of detail of the AES increases. Level 2 AES (AES-2) are undeniably leading to the best vector representations of programs.

In order to take into account the word/token order, *code*2*aes*2*vec* and *doc*2*vec* have been set up so far with concatenation as aggregation step. In order to confirm the importance of the order, we compare in Figure 5 embeddings obtained by the *code*2*aes*2*vec* algorithm with both types of aggregation (sum vs. concatenation). Unlike for concatenation, we observe a very rapid degradation in the quality of the embeddings obtained with a sum type aggregation when the size of the context increases. Indeed, the vocabulary on which AESs are based is very limited (only a few dozen or even hundreds of tokens) and the distribution of these words in AESs is not uniform. Thus it quickly becomes difficult to differentiate contexts as their size increases without taking into account the word order.

4.3 Application to feedback propagation

In order to confirm that the learned embeddings are fine enough to be usefully exploited in an educational context, we have implemented a first proof of concept on the task of propagating feedbacks.

We have considered the exercise *'mean'* from the dataset **NewCaledonia-5690** whose instruction was to write a Python

[5]The other methods presented in the state of the art could not be compared because of the lack of available operational implementations.

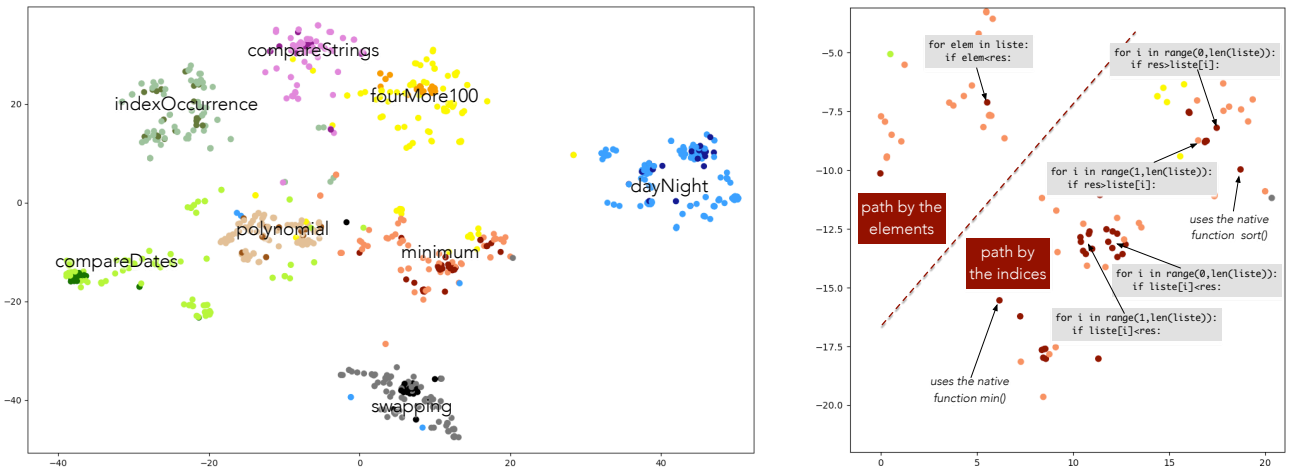

Figure 4: Visualization of program embeddings obtained by the method *code2aes2vec* for the 8 exercises of the dataset *NewCaledonia-1014*. The colors identify the exercises, with incorrect programs in light and correct ones in dark. The figure on the left represents all the embeddings and the one on the right details the area associated with the 'minimum' exercise.

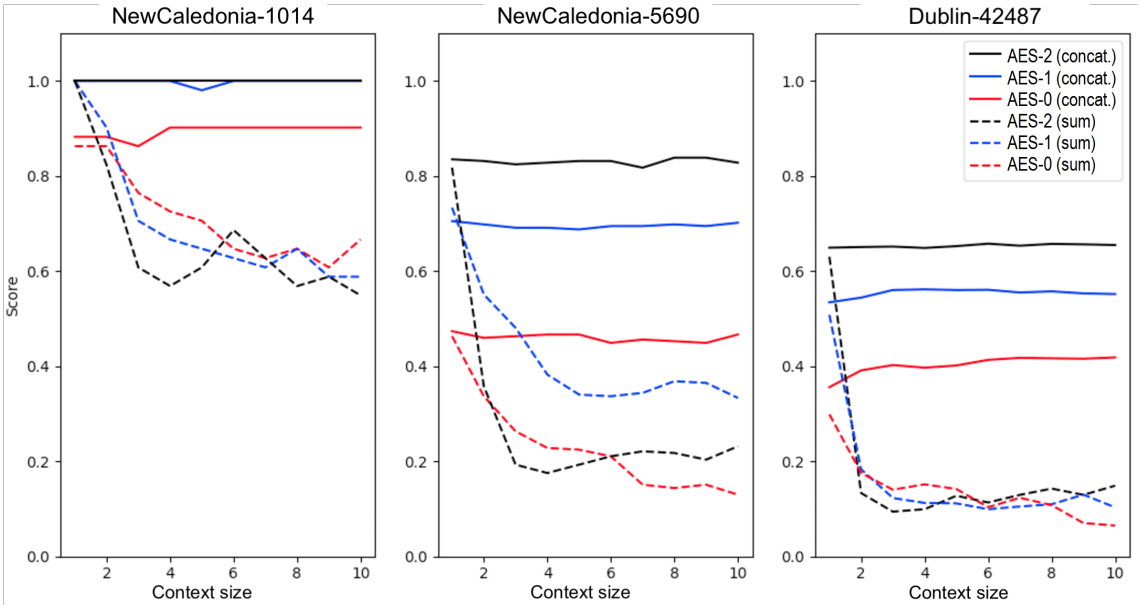

Figure 5: Evaluation of the embeddings on each of the three datasets according to the type of AES, the aggregation method and the context size (nb. words before/after).

function returning the mean of the values contained in a list passed as a parameter (and returning None if this list is empty). For this exercise, 157 programs were submitted to the platform by 24 different students ; among these submissions 122 programs were evaluated as incorrect by the platform and on which we sought to propagate teacher feedbacks based on their embeddings.

For this purpose, we performed a clustering (k-means [10]) on the 122 incorrect programs. We then presented to a teacher the most representative program (cluster medoid) of each cluster obtained, asking him to provide one feedback to help the student correct his proposal[6]. Once the k feedbacks were compiled (one per cluster/medoid), we went through each cluster and asked the teacher, for each program (other than the medoid), to indicate whether the feedback defined for the medoid could be applied to that other program. The objective is thus to assess the extent to which feedback from one medoid can propagate to all other programs in the same cluster.

Operationally, clustering is performed on the 122 incorrect

[6]If more than one error is found, the teacher must choose the one he/she feels needs to be corrected first. Each feedback is thus limited to the resolution of a single error.

programs, defined by their embeddings in $\mathbb{R}^{100}$. For a fixed number of k clusters, the partition selected to be analyzed is the one minimizing the MSE (Mean Square Error) among 100 runs (random initializations) of the k-means.

Table 4 presents the partition obtained for 5 clusters ($k = 5$). For each cluster we indicate its size, the program associated with its medoid as well as the feedback provided by the teacher for this medoid.

Let us first observe that the feedbacks provided by the teacher may relate to errors different in nature. It can be either an error in the design of the algorithm (clusters 1 and 3) or an error in the writing of the Python program (clusters 2, 4 and 5).

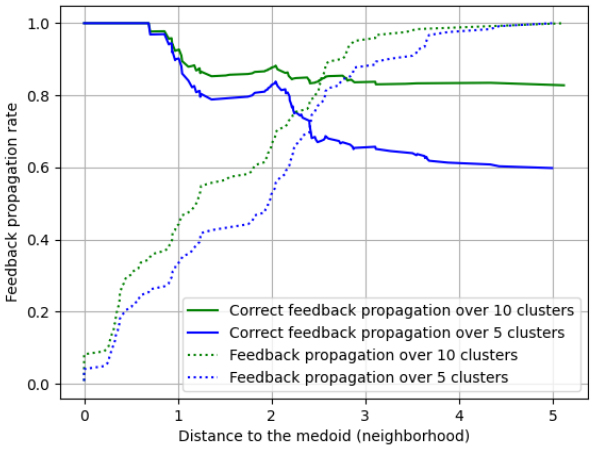

Figure 6: Propagation of teacher feedback to neighboring programs in each cluster. Illustration on the exercise *mean*.

We repeated this work with a partition into 10 clusters, this time asking the teacher to provide 10 feedbacks. Finally, we measured the rate of correct feedback when propagating the feedback from each medoid to its neighboring programs in the same cluster. Figure 6 shows the evolution of the correct feedback rate as a function of the neighborhood size considered for propagation. It can be seen that the further away from the medoids, the more errors in the automatically determined feedbacks.

Of course, a small number of feedbacks (5 or 10) is not enough to cover all the errors present in the 122 incorrect programs. In practice, it will therefore not be envisaged to propagate to all the programs in the cluster but only to the neighboring programs of the medoid. The dashed curves in Figure 6 indicate the proportion of programs covered as a function of the size of the neighborhood under consideration. We can see that with a neighborhood radius corresponding to a distance of 1.0, a propagation over 5 clusters allows to cover 33% of the programs with a precision of 90%; similarly such a propagation on 10 clusters allows to cover significantly more programs (43%) with a higher precision (93%).

The use of embeddings allows in this example to assist the teacher in his task of accompanying the students. For each feedback requested, 4 to 5 additional neighboring programs are automatically processed. Moreover, it is reasonable to think that over time a sufficiently large collection of feedbacks will be defined by the teacher to cover almost the entire embedding space so as to systematically identify one relevant feedback each time a new incorrect program is submitted and its embedding will position it in a pre-identified neighborhood.

5. CONCLUSION AND PERSPECTIVES

This paper studies the problem of learning vector representations, or embeddings, of programs in an educational context where the function is just as important as the style. Faced with this problem, we propose the method *code2aes2vec* transforming the code into abstract execution sequences (AES), and then into embeddings. This approach adapts the *doc2vec* method for program application and is based on the *document-based distributional variations* hypothesis.

The publication of the source code of the approach is accompanied by the availability to the community of a new enriched 'corpus' composed of more than 5,000 student programs (Python). Experiments conducted on these new data and on a public data set validate the quality of the learned embeddings, capturing in a fine way the function and the style of the programs. In addition a promising proof of concept was carried out on a classical task in the field, namely the propagation of teacher feedbacks.

The perspectives of this work are numerous. First, our experimentation focus on programs done in introductory courses, i.e. pretty simple codes. It would be interesting to analyze more elaborated ones (from more advanced courses) and to evaluate the impact of code complexity on performance.

Then, it seems necessary to complete the results observed on the stylistic differentiation of programs, by formalizing the notion of *style* of a program, in order to quantitatively evaluate our program embeddings. In the same way, a more precise analysis of the test cases used will have to be carried out in order to determine to what extent the constructed embeddings are sensitive to them.

Finally, to have more exploitable corpora, we plan to extend our implementation to handle any type of language (the current implementation only processes the Python language).

From a more methodological perspective, all the words in the program have the same weight during the embedding construction in our approach. Thus, a correct program and one returning a wrong value (or throwing an error) may have very similar embeddings, although functionally very different. This aspect could be integrated in the construction of our AES or in the architecture of the neural network used to generate embeddings. For that, it could also be interesting to add to our AES the values taken by the variables, in the same way that [17] but in a generic multimodal approach. Another perspective would be to allow the expert to integrate part of his knowledge on the language. As discussed previously, some instruction sequences can be equivalent

Cluster 1 (#31)	**Cluster 2** (#22)	**Cluster 3** (#28)	**Cluster 4** (#9)	**Cluster 5** (#32)

`def mean(l):`
`  if len(l)==0:`
`  res=None`
`  else:`
`  res=0`
`  cpt=0`
`  for elem in l:`
`  res=res+elem`
`  cpt=cpt+1`
`  res=elem/cpt`
`  return res`
	`def mean(l):`
	`  if len(l)==0:`
	`  res=None`
	`  else:`
	`  s=0`
	`  for elem in l:`
	`  s+=elem`
	`  res=s//len(l)`
	`  return res`
		`def mean(l):`
		`  if len(l)==0:`
		`  res=None`
		`  else:`
		`  res=0`
		`  cpt=0`
		`  avg=0`
		`  for elem in l:`
		`  res=res+elem`
		`  cpt=cpt+1`
		`  avg=res/cpt`
		`  return avg`
			`def mean(l):`
			`  if l==():`
			`  res=none`
			`  else:`
			`  res=0`
			`  for i in range(len(l)):`
			`  x=res+l[i]`
			`  res=x/len(l)`
			`  return res`
				`def mean(l):`
				`  if len(l)==0:`
				`  res=None`
				`  else:`
				`  res=0`
				`  cpt=0`
				`  for elem in l:`
				`  res=res+elem`
				`  cpt=cpt+2`
				`  res=res%cpt`
				`  return res`

The division step must be performed once the sum calculation is completed (put this instruction out of the `for` loop).	*The `//` operator corresponds in Python to the integer division. For the computation of a mean a simple division is required (operator `/`).*	*In the case of an empty list, your function does not return None (as requested).*	*The null value in Python is written 'None' (instead of 'none').*	*The `%` operator corresponds in Python to the modulus. For the computation of a mean a simple division is required (operator `/`).*

Table 4: Description of the 5 cluster partition generated by k-means on the embeddings of the incorrect programs from the *mean* exercise. For each cluster (table column): (1) the number of programs, (2) the program associated with the medoid of the cluster and (3) the feedback defined by the teacher for this program. Instructions in red are the ones that are questioned in the feedback.

(e.g., `if liste[i] <res:` vs. `if res> liste[i]:`). Semantic relations between words can also be known (e.g., the relation between `for` and `while` statements). This knowledge could be used to constrain the neural network and guide embedding construction. Finally, these program embeddings open up a large number of perspectives for teaching aid, in addition to the task of feedback propagation. For example, they could be used to identify error typologies, alternative solutions, or even predict dropout students through the analysis of their 'trajectories'.

6. 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*, 51(4):1–37, 2018.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the International Conference on Learning Analytics & Knowledge*, pages 86–95, 2019.

[4] R. Bazzocchi, M. Flemming, and L. Zhang. Analyzing cs1 student code using code embeddings. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1293–1293, 2020.

[5] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[6] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

[7] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

[8] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[9] M. LJPvd and G. Hinton. Visualizing high-dimensional data using t-sne. *J Mach Learn Res*, 9:2579–2605, 2008.

[10] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

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

[12] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring api embedding for api usages and applications. In *IEEE/ACM International Conference on Software Engineering*, pages 438–449. IEEE, 2017.

[13] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[14] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[15] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, page 1093–1102, 2015.

[16] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[17] K. Wang, R. Singh, and Z. Su. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*, 2018.