From document to program embeddings: can distributional hypothesis really be used on programming languages?

Thibaut Martinet, Guillaume Cleuziou, Matthieu Exbrayat, Frédéric Flouvat

From document to program embeddings: can distributional hypothesis really be used on programming languages?

Thibaut Martinet[a], Guillaume Cleuziou[a], Matthieu Exbrayat[a] et Frédéric Flouvat[b]

[a]University of Orléans, INSA-CVL, LIFO, EA 4022, F45067 Orléans, France
[b]Aix Marseille Univ, CNRS, LIS, Marseille, France

Abstract

Programming language processing is a field of increasing interest, as more and more models become available, either to address specific tasks or to acquire general knowledge which can then be fine-tuned on downstream tasks. All these models are based on architectures that come from the field of natural language processing, most of them being built on the distributional hypothesis from linguistics. Although this transition from one field to another appears to have occurred naturally, it is not so obvious to claim that this hypothesis will be appropriate for extracting semantics from programs.

In this paper, we investigate to which extent a distributional hypothesis can be applied to code embedding. To this end, we first formulate various hypotheses adapted to the specific information contained in programming languages. We then provide a framework to evaluate the effectiveness of these hypotheses through the quality of the resulting embedding spaces. This framework is based on the doc2vec model as a generic language model, as its implementation of the original distributional hypothesis is easy to understand and to adapt to any new ones. Among other tools, we propose a new evaluation method based on program analogies, which measures how well the models capture the underlying structure and meaning of the code.

We apply the proposed framework to a set of (distributional) hypotheses and show that we can rule out certain hypotheses in favor of others. Specifically, our study indicates that instruction-based hypotheses capture less semantic information than token-based ones. Furthermore, we observe that distributional hypotheses on tokens are effective in both source code, execution traces, and abstract syntax trees. Additionally, we find that the semantics captured on programs by these three hypotheses are of comparable levels and natures.

1 Introduction

The analysis of computer programs has been the subject of intense research, particularly in the last decade thanks to major advances in the fields of representation learning and deep learning. Methods for learning representations of programs and the processing models that derive from them are mainly applied in two areas: Software Engineering and Education (particularly in Computer Science Education).

In Software Engineering, these methods are used for tasks such as development assistance, debugging, refactoring, testing, etc. They help developers to improve the quality and reliability of their code, and can facilitate the maintenance and evolution of software systems.

In Education, these methods are used to assist students in learning programming and to help teachers track and analyze student learning patterns . They can provide personalized feedback and recommendations to students, and help teachers identify areas where students are struggling and adapt their teaching accordingly.

Many current models for processing computer programs are inspired by language models that were originally designed for natural language processing [20, 12, 46, 8]. The results obtained with these natural language models, for example on tasks such as code classification, generation, summarization, or translation, seem to suggest that the assumptions made about natural language can be transposed to programming languages. But is this always the case?

In this paper, we focus in particular on the distributional hypothesis of word meaning [15], which states that words with similar meanings appear in similar contexts. This hypothesis is indeed the basis for language models such as word2vec [23] and BERT [10], which have been widely used to process computer programs. However, it is not so obvious to claim that *similar fragments of computer code appear in similar contexts*. At the very least, it would be necessary to evaluate the effectiveness of this hypothesis with respect to the definition of such *fragments* and *contexts* in which they appear.

The aim of this paper is to investigate, independently of any specific downstream task, the extent to which *a* distributional hypothesis can be applied to programming languages in order to capture semantics. To our knowledge, this fundamental question has not yet been explored in these terms, due to two significant limitations: (1) the requirement for a language model that is centered on the distributional hypothesis and sufficiently generic

to instantiate multiple variants, and (2) the challenge of evaluating the quality of a task-agnostic embedding space, particularly in the case of program embeddings.

We have selected the doc2vec model [21] as a generic language model, given its ability to simultaneously learn continuous spaces of word and document representations by making simple and exclusive use of word distributions in documents. Our aim is to adapt this model to the specific context of programming languages, in order to compare different distributional hypotheses commonly encountered in this domain. We thus propose a generic reformulation of the doc2vec model that takes into account various definitions of *code fragment* and *context*. This enables us to learn continuous representations of code fragments (embeddings) and evaluate the effectiveness of different distributional assumptions in capturing semantics from computer programs.

We then choose to evaluate the effectiveness of these distributional hypotheses through the quality of the induced representation spaces. To this end, we have designed a framework for evaluating program embeddings during the learning process, based on several complementary perspectives: the visualization of program embeddings, the ability to partition according to an external criterion, and the capture of analogies.

Our framework allows one to assess the quality of program embeddings from different angles, providing a comprehensive evaluation of the effectiveness of the distributional hypotheses. By visualizing the embeddings, we can gain insights into the structure of the representation space and identify potential patterns or clusters. The ability to partition the embeddings according to an external criterion allows one to evaluate the extent to which the learned representations capture relevant information. Finally, by analyzing the syntactic-semantic relationships between programs, we can assess the extent to which the embeddings capture the underlying structure and meaning of the code.

The main contributions of this article are as follows:

- A synthesis and critical review of the different assumptions inherited from natural language and commonly accepted in programming languages.
- A generic reformulation of the doc2vec model to evaluate different scenarios of distributional hypotheses on computer programs.
- A task-agnostic and multi-perspective evaluation framework, including the novel construction of a set of over a thousand analogies on computer programs.
- A study highlighting the effectiveness of certain distributional hypotheses over others, demonstrated on several program datasets and confirmed over different languages (Python and Java).

2 Related work

2.1 Program embedding learning models

Concerning the transfer of natural language processing (NLP) methods to program analysis, the most intuitive way consists in observing the distribution of tokens (or sub-tokens [37]) in source code, *i.e.* to adapt the distributional hypothesis from natural language (NL) to programming language (PL) by mapping the words to tokens [12, 20, 13, 46, 26, 45, 30, 17]. A large majority of these models rely on the general Transformer architecture [39] or on one of its derivatives, such as BERT [10].

For instance, Kanade et al. [20] train a BERT model by adapting the original learning tasks to source code, *i.e.* the BERT NL sentences are mapped to PL statements. Neelakantan et al. [26] train a Transformer-based model in a contrastive learning way.

Some works also take advantage of comments, combining the analyses of tokens in source code and words in comments. Feng et al. [12] train a BERT model this way, as well as Wang et al. [46], Park et al. [30], who both train a Transformer-based encoder-decoder model with various PL-only and PL+NL tasks (*e.g.* generating comments from code).

Other works go even further. For example, Guo et al. [13] train a BERT model on several tasks including the prediction of some additional information from the data flow graphs. Wang et al. [44] train a BERT model with additional information from the abstract syntax tree. Wang et al. [45] train a Transformer-based model with additional inputs like AST and control flow graph.

Another approach consists in pre-processing source codes into an intermediate program representation (PR), the most common one being the abstract syntax tree (AST) [24, 25, 6, 47, 2, 41, 27, 8, 7, 28, 14, 45, 19]. Here, the models will observe the distribution of AST nodes, whether they correspond to tokens, instructions, or any other code fragment granularity.

For example, Mou et al. [25] adapt the convolution mechanism to tree structures, and apply it to AST to learn program embeddings. Their model has since been used in several works: Bui et al. [6] use it to learn program embeddings in a contrastive learning way; Bui et al. [7] add a layer that predicts the presence of some sub-trees in AST based on its embedding.

Guo et al. [14] propose a Transformer-based model that takes NL comments of the program combined with its AST that has been flattened in a way that keeps structural information.

Graphs are also commonly used as a PR that contains structural information of programs [1, 9, 4, 43, 13, 48, 45, 19]. It can either be AST augmented with additional information, or a flow graph, *e.g.* control-flow graph (CFG), or data-flow graph (DFG). In either case, the models will analyze the distribution of graph nodes, regardless of the code fragment granularity to which they correspond, as in AST.

Ben-Nun et al. [4] use a PR from a compiler tool (*i.e.* LLVM) to build a flow graph which combines information related to both CFG and DFG. They then train a word2vec model on the nodes, building the contexts as the set of neighbors in the graph.

Zhang et al. [48] propose to augment AST, *e.g.* by adding edges between close tokens in the source code or between siblings in AST, and to pass it through a variant of the Transformer architecture, adapted to graph structures.

Finally, another major approach is to execute the programs on test cases to extract specific information about their running behavior (or a derivative) [32, 42, 16, 40, 41, 8, 18, 17]. Here the models will analyze quite different distributions along the program executions, from the value of variables to the distribution of code fragments. For instance, Henkel et al. [16] apply word2vec on the instructions in the sequence of generated artificial traces, and Cleuziou and Flouvat [8] adapt doc2vec to analyze the AST nodes distribution in the sequence of traversed nodes during execution.

Beside the methods that explicitly rely on a distributional hypothesis, there are other approaches to learn program embeddings based on AST [47, 2, 28], graphs [1], or program execution [32, 42, 40, 41]. However, we will not discuss these approaches here as they do not observe a distribution in the same manner as the distributional hypothesis-based methods.

To sum up, there are four main categories of models that consider a distribution based on a context, as originally formulated in the distributional hypothesis, but applied to code fragments. These categories include source code-based models, AST-based models, graph-based models, and program execution-based models. Furthermore, some of these categories can be further divided into sub-categories that observe code fragments relative to PL tokens, instructions, or a combination of both.

2.2 Program embeddings evaluations

There exist only few genuine evaluations of embeddings in literature as most contributions focus on evaluating models through downstream tasks, which demonstrate their learning abilities rather than the quality of the embeddings themselves.

For example, a typical downstream task is to categorize programs, which can take the form of predicting or generating method names, exercises, problems, or any other predefined partition [24, 25, 47, 2, 40, 41, 7, 48, 19]. Another very common downstream task is to predict whether two programs are clones or to find the most likely clone of a program from a set of candidates [6, 47, 43, 13, 46, 44, 7, 14, 45, 19, 17]. Among the generative models, such as those based on the transformer architecture, some of their most common tasks involve converting NL comments or descriptions to PL code, the reverse, or even translating a program from one PL to another [12, 13, 46, 44, 7, 26, 14, 45, 30, 17].

In this paper, our focus is on evaluating the extent to which the model has successfully captured semantic information of programs into the embedding space. Few spatial evaluations go this way: Henkel et al. [16], Ben-Nun et al. [4] evaluate the embedding space through instructions analogies, Alon et al. [2] rather use function name analogies, and Cleuziou and Flouvat [8] propagate feedbacks between nearby programs in the embedding space.

Another method for evaluating code embeddings is to apply an external model to analyze them automatically. For instance, DeFreez et al. [9] use k-means clustering on program embeddings and compare the resulting clusters to given classes. Ben-Nun et al. [4], Cleuziou and Flouvat [8] use a recurrent neural network (RNN) and a support vector machine (SVM), respectively, to predict the exercise associated with a given code snippet. Bui et al. [7] apply k-means and evaluate whether similar programs tend to be grouped together in the same cluster. These approaches provide an automated way to assess the quality of the learned embeddings without relying on manual inspection or downstream task performance.

3 Distributional hypotheses formalization

Our study is focused on a representative subset of the different distributional hypotheses implemented by previous works. We consider six configurations of hypotheses, characterized through two dimensions: (1) the *code fragments* (or elements) whose semantic is targeted by the hypothesis and (2) the program representations (PR) specifying the *contexts of occurrence* in which the *elements* are distributed. In the following, two types of *elements*

are considered: tokens (t) and instructions (i); and three PR are studied: raw source code (S), execution trace (T) and abstract syntax tree (A).

We denote $\mathcal{H}_P^e$ the distributional hypothesis which states that *elements of type e with similar meanings appear in similar contexts in PR format P*. Table 1 summarizes the notations used for the six hypothesis configurations studied.

Table 1: The 6 distributional hypotheses studied on programming languages. Each hypothesis is defined by both, the *element* whose semantic has to be captured (lines) and the program representation specifying the contexts of occurrence in which the elements are distributed (columns).

	Source code (S)	**Execution trace** (T)	**Abstract syntax tree** (A)
Tokens (t)	$\mathcal{H}_S^t$	$\mathcal{H}_T^t$	$\mathcal{H}_A^t$
Instructions (i)	$\mathcal{H}_S^i$	$\mathcal{H}_T^i$	$\mathcal{H}_A^i$

4 Language model reformulation

To investigate the various distributional hypotheses in PL, our study relies on the word2vec language model, specifically its doc2vec extension. The word2vec language model [23] is formalized by a very light neural network trained on sequences of words (raw texts) in a self-supervised manner. In particular, in a sequence of words, the CBOW (Continuous Bag of Words) architecture predicts a hidden word from its context, *i.e.* the preceding and following words. The parameters of word2vec are summarized by two weight matrices: the matrix of word representations (or embeddings) and the interpretation matrix for predictions. In contrast, the SG (Skip-Gram) variant predicts the hidden context of a given word from the word itself.

Considering that word distributions vary from one text to another, Le and Mikolov [21] added the sequence in which the words are observed as an additional input. This results in the doc2vec model, which allows to learn both word and document (word sequence) embeddings simultaneously. Among the two versions of doc2vec presented in the original paper, we consider the PV-DM (Paragraph Vector - Distributed Memory) architecture, illustrated in fig. 1 (left), which corresponds to the augmentation of the CBOW architecture.

It is clear that these very light language models explicitly and exclusively rely on the distributional hypothesis of words in NL. Other language models such as transformers are indeed more powerful but are also more complex because they rely on other complementary hypotheses due to, for example, attention mechanism. More generally, we can reformulate doc2vec as a "semantic extraction tool" that leverages a distributional hypothesis on a set of structured data.

Let $\mathcal{E}$ be a set of *elements* (*e.g.* a set of words), $\mathcal{S}$ a set of structured data *sources* on $\mathcal{E}$ (*e.g.* raw texts = sequences of words), and $\mathcal{C}$ a definition of *context* on the occurrences of the elements of $\mathcal{E}$ in $\mathcal{S}$ (for example, the k preceding and following words). We denote $(\mathcal{S}, \mathcal{E}, \mathcal{C})$2vec, or sec2vec, the semantic extraction model that learns

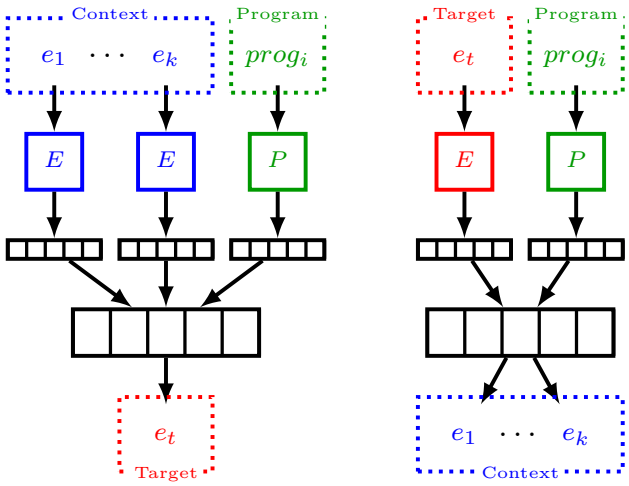

Figure 1: Sec2vec architectures: (left) PV-DM is trained to predict the target element from its context and the program; (right) PV-SG is trained to predict the context from its target element and the program.

embeddings on $\mathcal{S}$ (and $\mathcal{E}$) by analyzing the distributions of elements $\mathcal{E}$ in contexts $\mathcal{C}$ extracted from sources $\mathcal{S}$ (distributional hypothesis).

We then instantiate the sec2vec generic model on each of the six previously stated distributional hypotheses on PL, defining six different $(\mathcal{S}, \mathcal{E}, \mathcal{C})$ triplets (table 2).

Table 2: Definition of the triplets instantiating the six PL distributional hypotheses in sec2vec.

Distributional hypothesis	**Structured data sources** $\mathcal{S}$	**Elements** $\mathcal{E}$	**Contexts** $\mathcal{C}$
$\mathcal{H}_S^t$	raw source codes S (sequences)	tokens t	preceding and following elements in the sequence
$\mathcal{H}_S^i$		instructions i	
$\mathcal{H}_T^t$	execution traces T (sequences)	tokens t	
$\mathcal{H}_T^i$		instructions i	
$\mathcal{H}_A^t$	token-based ASTs	nodes in the tree	child nodes in the tree
$\mathcal{H}_A^i$	instruction-based ASTs		

Context definition depends on the kind of data structure. Source codes and execution traces are sequential, allowing contexts to be similar to those in natural language, which consist of the preceding and following elements. About AST, we take advantage of tree structures by using child nodes as the context of an element (*i.e.* a node).

Both tokens and instructions can be extracted naturally from source codes and execution traces with simple tokenization tricks. For ASTs, it is necessary to adjust the formalism by defining token-oriented and instruction-oriented AST structures (see Appendix[22]).

In the following, $Sec2vec(\mathcal{H}_P^e)$ denotes the sec2vec model applied on elements of type e in program representation P, taking as input the corresponding triplet $(\mathcal{S}, \mathcal{E}, \mathcal{C})$. Similarly, models based on the same type e of elements are denoted as $Sec2vec(\mathcal{H}^e)$, and on the same type P of program representations as $Sec2vec(\mathcal{H}_P)$.

We should mention the special case of token-based ASTs. In this PR, only leaves of the tree contain program tokens. As a result, since the $Sec2vec(\mathcal{H}_A^t)$ early version predicts each node from its children, it will never predict the leaf nodes and the program embeddings will never be directly learned from the tokens. To address this potential bias, we introduced a variant of doc2vec as shown in fig. 1 (right). We call this new version of doc2vec PV-SG since it is based on the word2vec SG architecture, and the sec2vec model using this architecture is denoted $Sec2vec^*(\mathcal{H})$.

In the following study, the $Sec2vec(\mathcal{H})$ model based on the PV-DM architecture will serve as our reference, and we will only report the performance of the PV-SG architecture ($Sec2vec^*(\mathcal{H})$ models) when it clearly benefits the quality of induced program embeddings (full comparative results are available in the Appendix[22]).

5 Datasets

To demonstrate our framework, we have collected three educational datasets of programs, whose overview is shown in table 3.

The first one, NC5690, has been proposed by Cleuziou and Flouvat [8], and consists of more than 5,000 students Python programs, organized into 66 exercises.[1] Most programs contain a single function, and none contain a class definition.

The two others, AD2022 and ProgPedia, have been proposed by Petersen-Frey et al. [31] and Paiva et al. [29] respectively, and both contain students programs in Python and Java. AD2022, the smallest one, contains less than a thousand programs in each language. Most of those programs only consist of a few functions, and rarely contain a class definition (even in Java), but they lack regularity since some of them contain several class definitions. Moreover, most of Python programs contain only a few functions. The ProgPedia dataset is made up of more complex programs, sometimes containing several class definitions, mainly in Java. This results in a richer "vocabulary".

The use of such educational datasets makes it possible to test distributional hypotheses on relatively simple programs naturally organized into distinct groups (in this case, exercises). This facilitates model analysis, offering both quantitative (e.g. measurements) and qualitative (e.g. visualizations) evaluation possibilities.

[1] Two programs are excluded from the original dataset for technical reasons.

Table 3: Some statistics on the datasets we use in this paper.

Dataset		**NC5690**	**AD2022**		**ProgPedia**	
Language		Python	Python	Java	Python	Java
Nb. programs		5690	687	838	2169	4339
Nb. exercises		66	21	21	7	16
Nb. students		56	166	199	187	254
Vocab. size	**# tokens**	94	152	121	154	245
	# instr.	1595	1455	1630	2857	6239
Av. prog. size	**# tokens**	65.14	125.15	174.8	503.05	533.5
	# instr.	9.79	15.97	17.33	69.06	57.16

6 Evaluation framework

This section presents one of the major contributions of the article, namely the evaluation of program embedding spaces. It aims to evaluate and compare the embeddings and through them the formulated distributional hypotheses, relying on external information or knowledge that the embedding spaces should have captured. By *external*, we mean that the models don't have access to them during their training phase. To our knowledge, there is no consensual methodology in the literature, and this is a real shortcoming for the domain.

Our set of evaluations starts with slightly revisited classical evaluations based on external information (*e.g.* a partition, classes, *etc*), followed by a new evaluation based on program analogies. To illustrate them, each evaluation has been applied on NC5690 dataset, although we obtained significantly similar results on other datasets (more detailed results are available in the appendix[22]).

Please note that not only we compare the models between different distributional hypotheses, but we also compare the same models with a different number of epochs, where one epoch is a complete pass through the entire dataset.

Every experiment has been run in a cross-validation fashion, where the datasets have been divided into 5 folds, and the models have been consecutively trained on 4 batches and evaluated on the fifth one so that each batch was tested once. Thus, in each table and curve, the values are averaged, and the standard deviation is indicated.

For the sake of reproducibility, we specify here the main parameters of the model: embedding vectors are 100-dimensional, the aggregation step consists of averaging the input vectors, the minimum number of element occurrences is 1 and context windows are composed of the 5 preceding/following elements (for sequential contexts). Variations have been tested, such as smaller window sizes or vectors, but no significant differences were observed.

6.1 Implementation

The sec2vec model is implemented in Python, using inner doc2vec models based on gensim [33] to learn program embeddings from a given corpus of program representations (PR). In order to compute the different PRs, we use the Python library tree-sitter [5], which contains parsers for a number of programming languages, and allows us to parse a piece of code and generate them. Please find examples of our AST and artificial trace in the appendix[22].Our models, evaluation functions and analogy dataset are available on github.[2]

The execution trace consists of the sequence of elements (*i.e.* tokens or instructions) traversed by the execution of the program. The underlying objective is to change the proximity of certain elements in the execution traversal order, thus modifying the raw source code distributional hypothesis with subtle additional information about the behavior of the program. To do so, we need to execute programs on test cases, but very few datasets contain test cases and it is a heavy task to build them manually from thousands of existing programs. Since our goal is to provide a framework which is easy to use, we decided to build artificial execution traces.

To generate an artificial trace, we implemented an algorithm that walks through the AST in a depth-first order, randomly excluding or repeating some sub-trees when specific nodes are reached. Sub-tree of a conditioned statement (*e.g.* if statement) has 50% chances of being excluded. When a loop is reached, its sub-tree is repeated a random number of times between 1 and 10. To explore as many program paths as possible, we actually concatenate 10 of these random walks to form a complete artificial trace. A comparison between the results obtained from the real and artificial trace showed that they were similar enough to proceed with the artificial trace.

Our goal is to capture the semantic of programs as algorithms, but a few aspects of source code slightly noise their semantic, namely NL-related semantic. For example, variable names for the same algorithm can be

[2]https://github.com/martinett/ProgramEmbeddingsEvaluationFramework

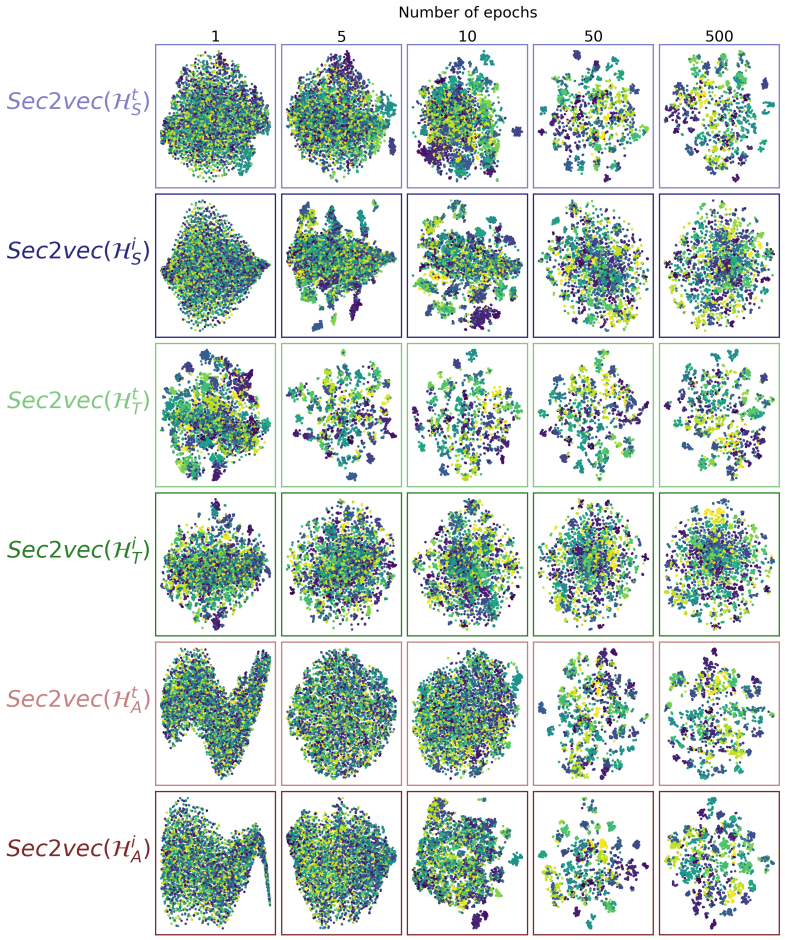

Figure 2: Cartography evolution on NC5690 from different distributional hypotheses, after training phases of different number of epochs, reduced to 2 dimensions by t-SNE models. The dot colors are based on the exercises.

very different between two programs, or the same name could correspond to very different variables, but the models will observe the distribution of those names. Another example is comments, which are often pure NL. So we decided to anonymize in all PRs every piece of code that holds potentially noising semantic for programs: function, class and variable names, comments, imported module and tool names, and constants. A more detailed description of the anonymization process is provided in the appendix[22].

6.2 Evaluations with external information

The first evaluations involve specific external information corresponding to semantic categories provided with the data. With NC5690, this information is the exercises answered by students.

We produce a first qualitative visualization, called a *cartography evolution*, to observe how the embedding spaces are structured during learning, depending on the distributional hypothesis considered, and thus gain initial insights. Each line in fig. 2 refers to a model (*i.e.* a sec2vec on a distributional hypothesis), each column corresponds to the duration of the training (*i.e.* the number of epochs: 1, 5, 10, 50 and 500), and each cell contains a 2-dimensional projection (using t-SNE [38]) of the program embeddings. The colors report the distribution of the external information.

First of all, a cartography evolution, in addition with its colors, allows us to see how much each model is effective in separating the different groups of a dataset in its embedding space. Figure 2 shows that in the first few epochs of the models, the embeddings seem to be almost uniformly distributed, and then start to form clusters as the number of epochs increases. This structuring mechanism suggests that a model is able to bring together programs from the same exercise, as long as the clusters are each of a single color. But we notice that *Sec2vec*($\mathcal{H}_S^i$) and *Sec2vec*($\mathcal{H}_T^i$), which basically analyze sequences of instructions, do not perform as well as the other models because the clustering process is much less prominent in this case. A possible cause for this behavior is that instructions are formed by concatenating their tokens, and it doesn't make a significant difference whether two objects differ by a large number of tokens or just a few, they will be analyzed and embedded totally independently. So these models could potentially give more importance to a difference of a few tokens between two programs than to a difference in structure. However, since the instructions-based models have to learn a much

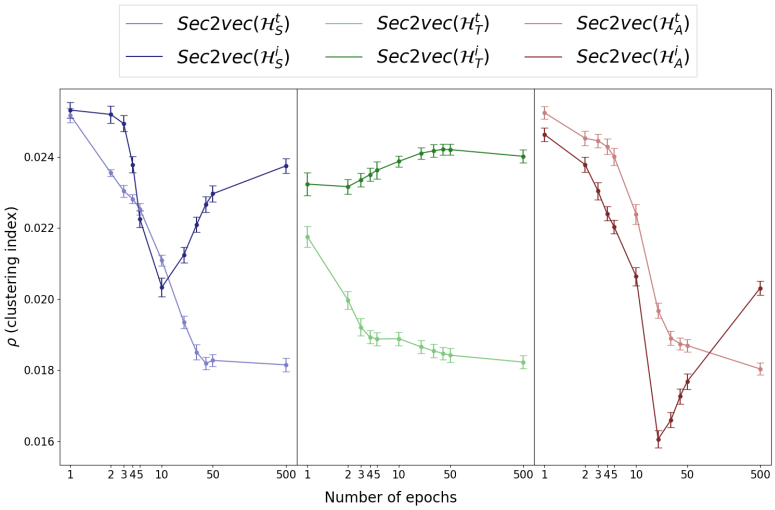

Figure 3: Clustering index ρ monitoring on dataset NC5690 the external information (here the exercises) captured by different sec2vec models: on source codes (blue), execution traces (green) and ASTs (brown).

Table 4: Examples of program analogies. Each row contains two pairs of programs (p_1, p_2) and (p_3, p_4), and we want that $\overrightarrow{p_1 p_2} \sim \overrightarrow{p_3 p_4}$.

	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$\mathbf{p_4}$
Syntactic	*for elem → for index*		*for elem → for index*	

	def sum_list(l):
	  res = 0
	  for elem in l:
	  res += elem
	  return res
		def sum_list(l):
		  res = 0
		  for i in range(len(l)):
		  res += l[i]
		  return res
			def display_list(l):
			  for elem in l:
			  print(elem)
				def display_list(l):
				  for i in range(len(l)):
				  print(l[i])

Semantic	*addition → multiplication*		*addition → multiplication*	

	def incr(x):
	  return x + 1
		def double(x):
		  return x * 2
			def incr_list(l):
			  return [e+1 for e in l]
				def double_of_list(l):
				  return [e*2 for e in l]

larger vocabulary than the token-based models, as shown in table 3, these two models might simply have not reached the clustered state yet. In contrast, $Sec2vec(\mathcal{H}_A^i)$ appears to perform well, likely because the distinction between similar structures is sufficient for the model to differentiate the exercises, and AST-based models like $Sec2vec(\mathcal{H}_A^i)$ are particularly effective at exploiting it.

Secondly, it gives us a glimpse of how fast each model structures the embedding space, in terms of number of epochs. For example, $Sec2vec(\mathcal{H}_T^t)$ and $Sec2vec(\mathcal{H}_T^i)$ achieve results in 1 or 5 epochs that are comparable to what $Sec2vec(\mathcal{H}_S^t)$ and $Sec2vec(\mathcal{H}_S^i)$ obtain in 10 or 50 epochs respectively, as if there is a 10-fold shift between these models. This is probably because the trace is actually a concatenation of 10 traces, each being quite similar to the source code. So not only do trace-based models have access to similar information as source code-based models, they also do similar work 10 times. This means that their weights are adjusted 10 times more frequently, allowing them to converge faster and achieve better performance in fewer epochs.

We supplement the visualizations with a quantitative assessment, in the form of a clustering index indicating how well program embeddings are separated into clusters of exercises (external information). The clustering index is defined as follows:

$$\rho(\mathcal{D}, \Pi_{\mathcal{D}}) = \frac{\sum_{\pi \in \Pi_{\mathcal{D}}} \sum_{p_i, p_j \in \pi} dist(p_i, p_j)}{\sum_{p_i, p_j \in \mathcal{D}} dist(p_i, p_j)} \tag{1}$$

where $\pi \in \Pi_{\mathcal{D}}$ are each class of the dataset external information, $p_i \in \mathcal{D}$ is a program of the dataset $\mathcal{D}$, and $dist(p_i, p_j)$ is the distance (typically euclidean distance) between the two programs in the embedding space. The clustering index $\rho \in [0, 1]$ is the ratio of the sum of intra-cluster distances to the total sum of distances over the entire dataset. We're not interested in the specific value of ρ (by definition always very small), but rather in how it changes and behaves during the process of learning embeddings. A decreasing value of ρ means that data from a same class are getting closer to each other. Conversely when ρ increases, data from a same class are dispersing.

This metric measures essentially the same information as the previous visualization, but it gives a quantitative value to compare models and avoids the bias of information loss inherent in the dimensionality reduction step

that is required for visualization. On the other hand, it suffers from another bias, since it penalizes a model that decomposes a class into several separated sub-clusters, even if this may be legitimate (*e.g.* several ways of solving an exercise). In any case, trends can be found between both evaluations, and they somehow complement each other. Figure 3 confirms the poor results of $Sec2vec(\mathcal{H}^i)$, except for $Sec2vec(\mathcal{H}^i_A)$ since it goes up towards the end. We also see the fast stabilization of $Sec2vec(\mathcal{H}_T)$ in just 4 or 5 epochs, where $Sec2vec(\mathcal{H}_S)$ stabilizes after 40 or 50 epochs (confirming the 10-fold shift previously mentioned).

The two main findings from this initial set of evaluations are as follows: firstly, analyzing the distributions of tokens (rather than instructions) seems to be the most effective way to extract semantics from programs; secondly, using execution traces would only artificially accelerate the learning of embeddings without providing any additional semantics compared to source codes or ASTs (since all $Sec2vec(\mathcal{H}^t)$ end up with approximately the same value).

6.3 Evaluation with program analogies

Table 5: Analogy evaluation per types, after 500 epochs on NC5690

	Source		Trace		AST	
	$Sec2vec(\mathcal{H}^t_S)$	$Sec2vec(\mathcal{H}^i_S)$	$Sec2vec(\mathcal{H}^t_T)$	$Sec2vec(\mathcal{H}^i_T)$	$Sec2vec(\mathcal{H}^t_A)$	$Sec2vec(\mathcal{H}^i_A)$
Syntactic analogies	**0.431** ±0.011	0.136 ±0.015	0.395 ±0.011	0.124 ±0.020	0.328 ±0.008	0.141 ±0.023
Semantic analogies	**0.961** ±0.005	0.001 ±0.002	0.955 ±0.007	0.0 ±0.0	0.551 ±0.043	0.008 ±0.006

The previous evaluations are interesting when one wants to compare several models on the same dataset, and when there is an available external information which organizes programs into semantically different classes. However, sometimes datasets do not contain this kind of information, and it would be costly in time and resources to manually design a partitioning on the dataset of programs. In addition, this kind of evaluation does not allow one to compare models on different datasets with language variations (*e.g.* Python *vs* Java) or domain variations (*e.g.* educational *vs* engineering dataset) since results are highly dependent on the number of classes.

This is why we propose a new evaluation based on program analogies, as well as large dataset of such analogies. To the best of our knowledge, this is the first attempt to create such a dataset, which is generic and can be used on any model that learns program embeddings. And yet, analogy evaluation seems to have proven its worth [11, 36], and is widely used in representation learning model evaluation [3, 35, 34, 23, 4, 2]. The difference is that we study analogies on *programs*, instead of the usual ones on *elements*.

Our analogy evaluation works in an usual manner: taking two pairs of connected programs (p_1, p_2) and (p_3, p_4), as illustrated in table 4, we infer the four embedding vectors $vec(p_i)$ and proceed to the following operation $p' = vec(p_2) - vec(p_1) + vec(p_3)$, which leads us to a point p' somewhere in the embedding space. Then, among all the dataset programs the model was trained on, we check whether p_4 is the nearest neighbor of p'. This is like applying the vector $vec(p_2) - vec(p_1)$ on $vec(p_3)$, expecting $vec(p_4)$ to be the resulting vector. This process will verify if the embedding space is coherent enough to retrieve the fourth program of each analogy.

As it is often the case, we define two types of program analogies:

- *Syntactic analogies:* leveraging syntactic relations between two programs semantically equivalent (produce the same output) but having different implementations.

- *Semantic analogies:* leveraging on semantic relations between two programs semantically different (differ on the outputs) but in a syntactically similar way.

The evaluation of program embeddings by analogies brings a new and very valuable dimension to the study on distributional hypotheses applied to programming languages. In order to evaluate as much embedding space as possible, we built a set of more than a thousand program analogies, similar to the examples in table 4, consisting of 67% syntactic analogies and 33% semantic analogies. We then computed the accuracy of each model on this set of analogies as the ratio of successfully retrieved analogies (*i.e.* analogies for which the target program (p_4) is the nearest neighbor of the resulting vector p').

Table 5 shows the accuracies for the six models (candidate distributional hypotheses) on both analogy types. This table confirms very bad results of instruction-based models. Token-based models are excellent for retrieving semantic analogies, but they do not perform as well with syntactic ones. Interestingly, regarding performance between syntactic and semantic analogies, we obtain the same trend as Di Gennaro et al. [11] who evaluate a word2vec with word analogies in NL. Note that Ben-Nun et al. [4], who evaluate a word2vec with instruction analogies in PL, obtain less differences between syntactic and semantic analogies, but their definition of these

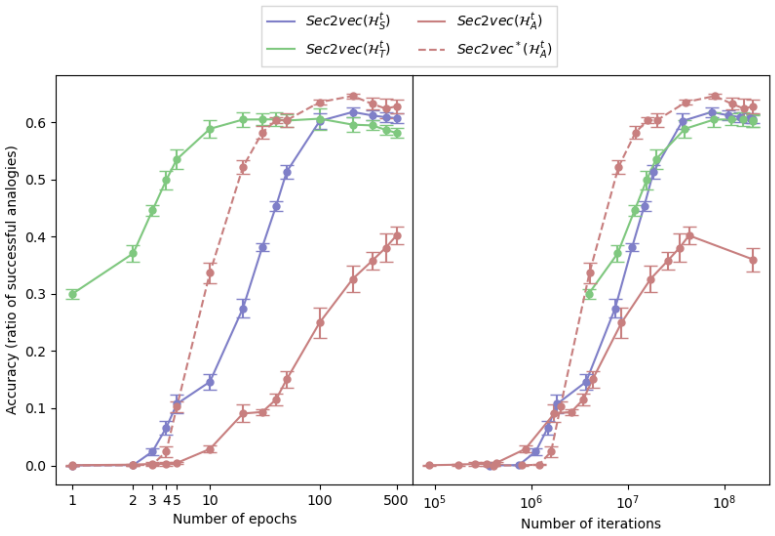

Figure 4: Analogy evaluation of different sec2vec models applied on NC5690, after training phases of different number of epochs, aligned by number of epochs on the left, and by number of iterations on the right (one epoch consists of as many iterations as learning examples) .

analogies are quite different. Table 5 also shows that $Sec2vec(\mathcal{H}_A^t)$ is not competitive, thus supporting our assertion made in section 4 that token-based ASTs are poorly exploited by a PV-DM architecture.

Table 6 reports results obtained with token-based models on all datasets. It definitely confirms that $Sec2vec(\mathcal{H}_A^t)$ is not competitive unless we include $Sec2vec^*(\mathcal{H}_A^t)$, which leverages the token-based ASTs with a skip-gram based architecture. Results even show that it is slightly better than other PRs, but not systematically nor significantly. Note that there are more results about other $Sec2vec^*(\mathcal{H})$ in the appendix[22]. These findings support the claim that execution traces do not appear to offer valuable supplementary information in comparison to source codes, as $Sec2vec(\mathcal{H}_T^t)$ rarely outperforms $Sec2vec(\mathcal{H}_S^t)$.

To confirm this definitely, we studied the evolution of the model performance according to the number of epochs. As illustrated in fig. 4 (left), $Sec2vec(\mathcal{H}_T^t)$ is the first one to stabilize, around epoch 10, while $Sec2vec(\mathcal{H}_S^t)$ stabilizes around epoch 100, so we have once again this 10-fold shift between them. $Sec2vec(\mathcal{H}_A^t)$, does not seem to have reached its limit yet at epoch 500.

Furthermore, we investigated the question about the number of iterations (one epoch consists of as many iterations as learning examples), and it turns out that the models indeed perform different number of predictions per epoch, favoring those computing more. For example, $Sec2vec(\mathcal{H}_T^t)$ has about 10 times more iterations per epoch than $Sec2vec(\mathcal{H}_S^t)$ on NC5690. To deal with this bias, we display in fig. 4 (right) how accuracy evolves with the number of iterations. A notable observation from the figure is that the gaps between the curves have largely disappeared. It seems that for the same number of iterations, all the models' accuracy increases almost simultaneously. Note that curves start at different values because the first point is actually the accuracy at the end of the first epoch, so the more a model has iterations per epoch the later its curve starts in the graphic. Also note that to obtain a satisfactory final comparison, we trained models once again on a specific number of epochs in order to make curves to end simultaneously, and the new accuracy value of $Sec2vec(\mathcal{H}_A^t)$ suggests that this model has finally reached its limit, albeit below the others. Thus, these graphics confirm that $Sec2vec(\mathcal{H}_A^t)$ is not as good as the three others, which are approximately equivalent according to the number of iterations.

7 Conclusion

In this paper, we formulated several distributional hypotheses for programming language, from the distribution analysis of tokens to instructions, depending on the contexts of program representations from source code to AST and execution trace. We have adapted a light language model, namely doc2vec, to be able to precisely study the impact of different distributional hypotheses on embeddings, and also proposed various evaluation strategies to assess and compare them. We applied these strategies to evaluate models for different numbers of epochs, and on several datasets in Python and Java. Languages do not drastically change the results, but it will be interesting to further investigate on the differences between them, or even on the possible benefits of training a multi-language model.

Among these evaluations, we proposed a new program analogy assessment solution, that allows to verify whether models have built a syntactically and semantically coherent embedding space. It is based on the addition

Table 6: Analogy evaluation per dataset, after 500 epochs

	Sec2vec()			*Sec2vec**()
	$\mathcal{H}_S^t$	$\mathcal{H}_T^t$	$\mathcal{H}_A^t$	$\mathcal{H}_A^t$
NC5690	0.607	0.581	0.402	**0.628**
AD2022 (Python)	**0.717**	0.658	0.423	0.669
AD2022 (Java)	0.695	0.751	0.433	**0.761**
ProgPedia (Python)	0.714	0.686	0.357	**0.761**
ProgPedia (Java)	**0.842**	0.829	0.516	0.772

of new programs into embedding space and analyzing their relative coordinates. We then proposed a set of more than a thousand analogies to have a maximum cover of the embedding space, but the proposed set of analogies is intended to be enriched to cover even more space and evaluate new program relations.

We found that the instruction-based distributional hypotheses fail to extract the semantics of programs. On the other hand, token-based distributional hypotheses seem to be globally equivalent.

As perspectives, we could explore a better AST format, to provide models with even more useful information about the program structures. It would also be interesting to augment these ASTs into graphs to further extend the current hypothesis list. In another direction, we also plan to analyze how our framework performs on deeper, not yet fine-tuned state-of-the-art models. Since the field of evaluating distributional hypotheses for programming language is still unexplored, there are plenty of possible directions to follow, and thus plenty of opportunities for future works. We hope that this paper will serve as a foundation for further investigations to consolidate the knowledge of this field, and maybe one day to improve or at least to better understand the bigger deep language models for programming language.

References

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[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] A. Bakarov. A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536*, 2018.

[4] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in neural information processing systems*, 31, 2018.

[5] M. Brunsfeld, P. Thomson, J. Vera, A. Hlynskyi, P. Turnbull, T. Clem, and A. Muller. tree-sitter/tree-sitter: v0. 20.0, 2018.

[6] N. D. Bui, L. Jiang, and Y. Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *Workshops at the AAAI Conference on Artificial Intelligence*, 2018.

[7] N. D. Bui, Y. Yu, and L. Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *ACM 43rd Int. Conference on Software Engineering (ICSE)*, pages 1186–1197. IEEE, 2021.

[8] G. Cleuziou and F. Flouvat. Learning student program embeddings using abstract execution traces. In *14th International Conference on Educational Data Mining*, pages 252–262, 2021.

[9] 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.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*, pages 4171–4186, June 2019. doi: 10.18653/v1/N19-1423.

[11] G. Di Gennaro, A. Buonanno, and F. A. Palmieri. Considerations about learning word2vec. *The Journal of Supercomputing*, pages 1–16, 2021.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[14] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[15] Z. S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954. doi: 10.1080/00437956.1954.11659520. URL `https://doi.org/10.1080/00437956.1954.11659520`.

[16] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

[17] J. Huang, J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen. Code representation pre-training with complements from program executions. *arXiv preprint arXiv:2309.09980*, 2023.

[18] E. Jabbar, S. Zangeneh, H. Hemmati, and R. Feldt. Test2vec: An execution trace embedding for test case prioritization. *arXiv preprint arXiv:2206.15428*, 2022.

[19] Y. Jiang, X. Su, C. Treude, and T. Wang. Hierarchical semantic-aware neural code representation. *Journal of Systems and Software*, 191:111355, 2022.

[20] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.

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

[22] T. Martinet, G. Cleuziou, M. Exbrayat, and F. Flouvat. Appendix of the paper "From document to program embeddings: can distributional hypothesis really be used on programming languages?", 2024. URL `https://hal.science/hal-04666404`.

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

[24] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*, 2014.

[25] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, 2016.

[26] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

[27] B. Paassen, I. Koprinska, and K. Yacef. Tree echo state autoencoders with grammars. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[28] B. Paaßen, I. Koprinska, and K. Yacef. Recursive tree grammar autoencoders. *Machine Learning*, 111(9):3393–3423, 2022.

[29] J. C. Paiva, J. P. Leal, and Á. Figueira. Progpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief*, 46:108887, 2023.

[30] Y. Park, A. Park, and C. Kim. Alsi-transformer: Transformer-based code comment generation with aligned lexical and syntactic information. *IEEE Access*, 2023.

[31] F. Petersen-Frey, M. Soll, L. Kobras, M. Johannsen, P. Kling, and C. Biemann. Dataset of student solutions to algorithm and data structure programming assignments. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 956–962, 2022.

[32] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*, pages 1093–1102. PMLR, 2015.

[33] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC Workshop on New Challenges for NLP Frameworks*, pages 45–50. ELRA, May 2010.

[34] A. Rogers, A. Drozd, and B. Li. The (too many) problems of analogical reasoning with word vectors. In *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (* SEM 2017)*, pages 135–148, 2017.

[35] J. Santos, B. Consoli, and R. Vieira. Word embedding evaluation in downstream tasks and semantic analogies. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 4828–4834, 2020.

[36] N. Schluter. The word analogy testing caveat. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Volume 2*, pages 242–246. Association for Computational Linguistics, 2018.

[37] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[38] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[40] K. Wang. Learning scalable and precise representation of program semantics. *arXiv preprint arXiv:1905.05251*, 2019.

[41] K. Wang and Z. Su. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 2020.

[42] K. Wang, R. Singh, and Z. Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[43] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[44] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021.

[45] X. Wang, Y. Wang, Y. Wan, J. Wang, P. Zhou, L. Li, H. Wu, and J. Liu. CODE-MVP: Learning to represent source code from multiple views with contrastive pre-training. In *Findings of the Association for Computational Linguistics*, pages 1066–1077, July 2022. doi: 10.18653/v1/2022.findings-naacl.80.

[46] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[47] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[48] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*, pages 378–389, 2022.