BEYOND SYNTAX TREES: LEARNING EMBEDDINGS OF CODE EDITS BY COMBINING MULTIPLE SOURCE REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning efficient distributed representations of *code edits* is fundamental for various software engineering tasks, such as the automatic identification of commits that introduce or correct vulnerabilities. Some successful models, including `commit2vec` and `edit2vec`, represent code changes using paths extracted from the Abstract Syntax Trees (AST). Other works have shown that, in addition to the AST, considering graph structures that encode the control flow and the data flow of a program can lead to more effective code embeddings.
In our work, we introduce a new model to represent code edits that leverages different paths derived from the AST, the Control Flow Graph (CFG) and the Data Flow Graph (DFG). Our preliminary evaluation on the task of classifying security-relevant commits yielded encouraging results that call for further investigation.

1 INTRODUCTION

Learning distributed representations of source code is the central problem of many deep-learning based approaches to solve software-engineering tasks such as code clone detection, code summarisation or vulnerability identification. There is now an already considerable and nevertheless rapidly growing number of works that address this problem (Alon et al., 2019b; Allamanis et al., 2018; Ma et al., 2022; Hellendoorn et al., 2020; Feng et al., 2020). Some of these works are inspired by models coming from the field of *Natural Language Processing (NLP)* (Feng et al., 2020; Lachaux et al., 2020). Others leverage the formal nature of programming languages and rely on *Abstract Syntax Trees (AST)* or other graph representations of code (Alon et al., 2019b;a; Hellendoorn et al., 2020; Allamanis et al., 2018), and finally some combine the two approaches (Guo et al., 2021; Zügner et al., 2021).

Compared to the literature on code representation methods, the volume of works that focus on learning a representation of *code edits* or "commits" is more modest (Cabrera Lozoya et al., 2021; Yin et al., 2019; Hoang et al., 2020; Qureshi et al., 2021). However, only in rare occasions the source code of a program remains static and unchanged for long periods of time. The process of software development is essentially incremental: most programs and libraries are continually updated. Consequently, often times, program analysis tasks require scanning several versions of the source code to identify when and where a particular change was introduced, and the impact it has on the program itself. An important example is identifying commits that introduce or correct vulnerabilities (Cabrera Lozoya et al., 2021).

Of the many different code representation models, only a few can be easily adapted to obtain distributed representations of code edits. Among these, `code2vec` is a the widely successful method that leverages the *context* (the actual code tokens) and *paths* extracted from its *Abstract Syntax Tree (AST)*. In `code2vec`, code is initially represented as a set of so-called *path-contexts* and subsequently embedded using an attention-based neural network.

code2vec constitutes the basic building block of the commit2vec model (Cabrera Lozoya et al., 2021). Its basic principles (initial representation based on paths extracted from the AST, attention-based network) are used in other embedding methods for code changes, such as edit2vec (Qureshi et al., 2021). Whether these path-based approaches lead to more effective representations with respect to simpler models is disputed (Qureshi et al., 2021), however they have proven to be at least as good as the baselines on several tasks, and are considered state of the art. Moreover, recent works (Vagavolu et al., 2021) have shown that path-based approaches like code2vec could be improved by leveraging the semantic information encoded in graph representations of the code, such as the *Control Flow Graph (CFG)* and the *Program Dependency Graph (PDG)*.

Building on the previous works of Cabrera Lozoya et al. (2021), and Vagavolu et al. (2021), we introduce a new path-based model for the distributed representation of code edits, that combines multiple representations of the code associated to different graph structures. In particular, we consider different types of paths extracted from the AST, the CFG and the *Data Flow Graph (DFG)*. We define a new method for extracting meaningful paths from the CFG that requires splitting the graph into its *basic blocks*. We show how this method leads to better computational efficiency and reduced redundancy, when considering changes in the control flow. Our model can be applied to any programming language, whereas the generation of the AST, CFG and DFG from source code is language-specific.

We evaluate our model on the task of classification of security-relevant commits in Java projects using the dataset from Ponta et al. (2019). Comparison with current state of the art (commit2vec) shows that considering multiple representations can improve the classification performance.

This is the layout of the paper: in the next section we introduce previous work on which our model is built upon. The third section is dedicated to the presentation of the model itself, with a focus on the innovative aspects with respect to previous works. In the fourth section we report some details on the practical implementation, and the results of the evaluation on the task of classification of security-relevant commits. Finally, the last section contains the concluding remarks.

2 BACKGROUND

The main idea of our model is to represent code-changes by combining multiple path-based representations obtained from different graph abstractions of the code. We follow a similar approach to the one used by Vagavolu et al. (2021) in their "mocktail" of source code representations. To apply this method to commits we employ the same methodology used in commit2vec, that allows to represent commits using the path-contexts introduced in code2vec. Differently to commit2vec, we we rely on the code2seq encoder to obtain the embeddings.

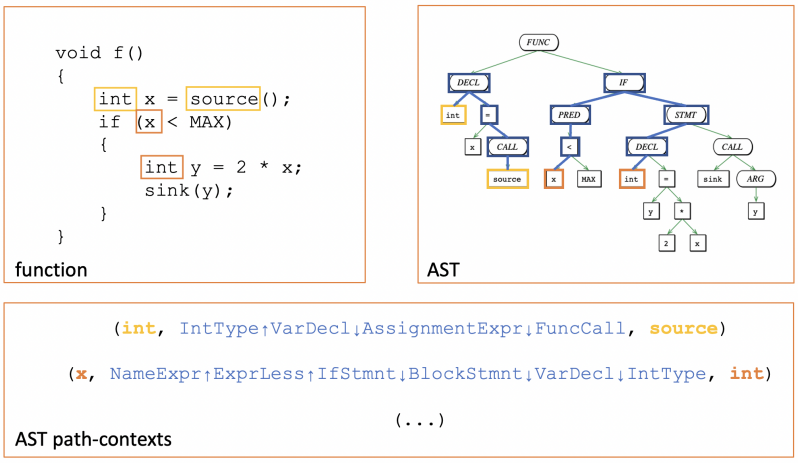

Figure 1: Example of two AST path-contexts used in code2vec and derived models.

code2vec & commit2vec. code2vec (Alon et al., 2019b) is an attention-based representation learning model for source code at the function (or method) level, that relies on an initial representation of the code as a set of path-contexts extracted form its AST. More formally, path-contexts are triplets of the form $(t_1,\ p,\ t_{k+1})$, where t_1 and t_{k+1} are the code tokens associated to two terminal nodes of the AST (the "start" terminal node n_1 and the "end" terminal node n_{k+1}), and p is the path in the AST connecting them. p is represented by a sequence of the form $l_1 d_1 l_2 d_2 ... l_k d_k l_{k+1}$ where $l_1, ..., l_{k+1}$ are the labels associated to the nodes $n_1, ..., n_{k+1}$ and $d_1, ..., d_k$ are the directions of movements between path nodes in the AST. The symbol $\uparrow$ represents upward movement and $\downarrow$ represents a downward movement in the tree. Figure 1 shows an example of two path contexts obtained from a simple AST. The terminals and the path of each path-context are embedded using a simple token embedding and concatenated to obtain a *path-context vector*. Features are extracted from each path-context vector using a fully-connected linear layer followed by a layer normalization and an activation function. Finally a global attention mechanism is used to aggregate the information from all the path-context vectors. A schema of the model is shown in Figure 2.

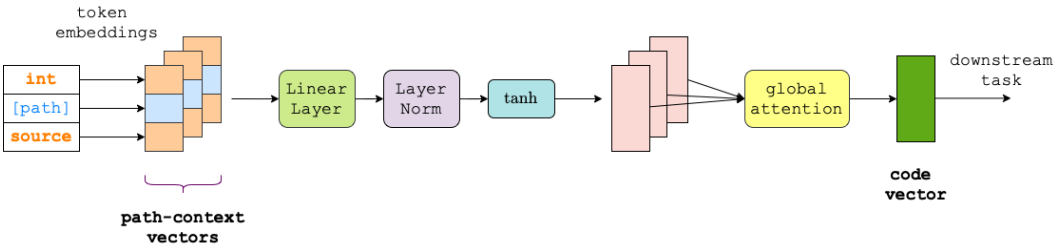

Figure 2: code2vec model.

Cabrera Lozoya et al. (2021) devised a way to represent commits as a set of path-contexts. In commit2vec, a commit C is defined as a change in the source code of a given project in a set of files $f_i \in \mathbb{F}$, where $i = 1, \ldots, I$, and where I is the number of files changed within C. The concept of commit implies a prior and a posterior version of files f_i, denoted respectively $f_{i,pre}$ and $f_{i,post}$. Let the set of all the path-contexts of the prior versions of all methods $m_{1..J,pre}$ in all files $f_{1..I,pre}$ in commit C be defined as $\mathcal{S}_{pre} = \{\pi_1, \pi_2, \ldots, \pi_k\}$, and the set of all the path-contexts of the posterior versions of all methods $m_{1..J,post}$ in all files $f_{1..I,post}$ in commit C be defined as $\mathcal{S}_{post} = \{\pi_1, \pi_2, \ldots, \pi_k\}$ (to avoid confusion, we use the letter π to indicate *path-contexts* as a whole, as opposed to the paths that are part of the path-contexts, indicated with the letter p). Then, the set of path-contexts describing a commit C is defined as the *symmetric difference* between $\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$:

$$\mathcal{S}_C = \mathcal{S}_{post} \Delta \mathcal{S}_{pre} = \{\pi : \pi \in \mathcal{S}_{pre} \cup \mathcal{S}_{post}, \pi \notin \mathcal{S}_{pre} \cap \mathcal{S}_{post}\}$$

$\mathcal{S}_C$ is the input provided to the code2vec neural network to obtain a distributed representation of the code changes contained in commit C.

A "Mocktail" of path-based code representations. Vagavolu et al. (2021) showed that one can improve the representation capabilities of code2vec by combining the original AST-based embedding with other representations of the code, obtained using paths derived from the CFG and the PDG. These graphs encode semantic information about the code that cannot be extrapolated from the AST alone. In order to do that, they defined new methods for extracting path-contexts from the DFG and the CFG. In particular, the paths in the PDG path-contexts represent sequential dependencies between statements, and are defined in a straightforward manner as all simple paths connecting *source nodes* (nodes with no incoming PDG edges) to *sink nodes* (nodes with no outgoing PDG edges). The contexts are the code tokens associated to the source and sink node. Path-contexts from the CFG are defined in a similar fashion as simple paths connecting source nodes to sink nodes. In this case, the source node is unique (the $Methods$ node n_1), and the sink nodes n_{k+1} can be either:

- The $Return$ node,
- A previously visited intermediate node that represents a loop control structure (i.e., $n_{k+1} \in \{n_2, \ldots, n_k\}$).

Each of the CFG paths represents a control flow during program execution. To represent loops in CFG, three different paths are extracted from it - the first one ignores the loop and proceeds to the next node, the second path goes through the loop only once and proceeds to the next node, the last one goes back to the visited loop node and ends there. The architecture proposed in (Vagavolu et al., 2021) consists of three stacked `code2vec` networks, each used to obtain an embedding of the path-contexts extracted from the different graphs. The final *code vector* is a concatenation of the three vectors obtained from the AST, CFG and PDG representations.

`code2seq`. Previous works use `code2vec` as a basic building block. Alon et al. (2019a) introduced a variant of the method called `code2seq`. As the name suggests, this is a `seq2seq` model that was originally applied to the task of code summarisation.

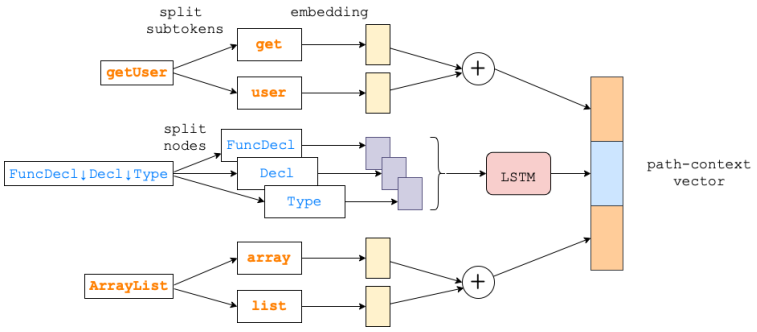

Figure 3: The `code2seq` path-context encoder.

The encoder part of `code2seq` proposes an alternative way to embed each individual path-context, in particular there are two main differences with respect to `code2vec`, that are shown in Figure 3:

- The terminals (code tokens) are first split into sub-token, and the embedding for the terminal is the sum of the embedding of each sub-token.
- Instead of using a simple token embedding, paths are considered as sequences of nodes and are embedded using a Long-Short Term Memory network (LSTM).

Thanks to these gimmicks, the `code2seq` encoder has fewer parameters with respect to `code2vec`. Moreover, it solves the problem of Out-Of-Vocabulary (OOV) paths, since they are represented as sequences of nodes (which in turn are fixed in number), and significantly mitigates the risk of encountering OOV contexts, thanks to the sub-token split.

3 Mocktail of Code Edit Representations

We propose a new multi-path approach inspired by the work of Vagavolu et al. (2021) to learn distributed representations of code changes. Following the approach of `commit2vec`, we initially represent a code edit using three different sets of symmetric path-context differences:

$$\mathcal{S}_C^{(*)} = \mathcal{S}_{post}^{(*)} \Delta \mathcal{S}_{pre}^{(*)} = \{p : \pi \in \mathcal{S}_{pre}^{(*)} \cup \mathcal{S}_{post}^{(*)}, \pi \in \mathcal{S}_{pre}^{(*)} \cap \mathcal{S}_{post}^{(*)}\},$$
$$\text{where } * \in \{AST,\ CFG,\ DFG\}$$

The main differences with respect to the work of (Vagavolu et al., 2021) are the following:

1. Instead of the Program Dependency Graph (PDG) we extract paths from the Data Flow Graph (DFG). DFGs specifically represent data dependencies between the different statements.
2. We propose an alternative way of extracting paths from CFG that we name *Control Flow Basic Blocks (CFBB),* which is more suited to represent code edits. Details about this method are presented next.

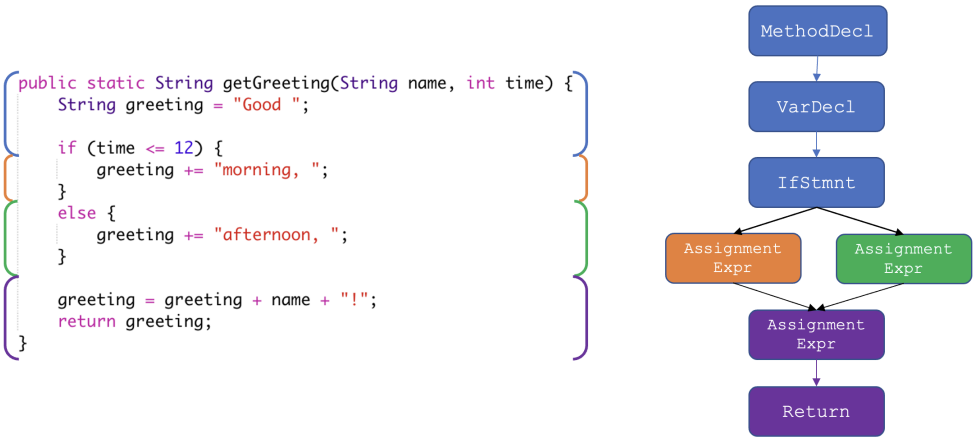

Figure 4: Example of Control Flow Basic Blocks. The graph on the right is a slightly simplified version of the CFG of the code on the left. Each of the four different colors represents separate building blocks.

Control Flow Basic Blocks (CFBB). In (Vagavolu et al., 2021) each CFG path represents a complete control flow during program execution. The number of possible CFG paths is thus exponential with respect to the number of control structures (i.e., if, for, or $while$ statements), since any of these introduce a bifurcation in the graph. In the original paper, this was not an issue since the model was applied to methods written in the C programming language. As stated by the authors, C functions tend to be shorter and have simpler control flows with respect to projects written in other programming languages like Java (Vagavolu et al., 2021). In the context of our work, this is a major computational bottleneck.

The specific issue we address is how to represent commits: we do not consider the path-contexts extracted from methods themselves, but the *difference* (Δ) of path-contexts between the prior and the posterior version of the method. The problem with complete control flows is that many parts or "blocks" of the control flow are included in all the CFG paths. If a change is made involving one of these blocks it can potentially modify all the CFG paths of the method (we empirically observe this is the case most of the time). In this scenario the set $\mathcal{S}_C^{(CFG)}$ will contain all the control flows from $\mathcal{S}_{pre}^{(CFG)}$ and $\mathcal{S}_{post}^{(CFG)}$ since $\mathcal{S}_{post}^{(CFG)} \cap \mathcal{S}_{pre}^{(CFG)} = \emptyset$. As a consequence, extracting the complete control flow not only can be impractical, but also leads to redundancy and noise in the data.

To tackle this issue, we propose to split the control flows into its basic blocks. A *Control Flow Basic Block (CFBB)* is a sequence of nodes connected by a CFG edge. Segments terminate when one of these conditions is met:

1. A terminal node is reached $(Return)$.
2. A bifurcation is encountered. Then, the next nodes in the bifurcation become the starting point of new separate segments.
3. A node with more than one incoming CFG edge is reached. Then, the following node(s) become the starting point(s) of a segment.

Basic blocks are the fundamental units that can form every possible complete control flow from the method definition to the $return$ statement(s). An example of how Control Flow Basic Blocks is shown in Figure 4.

Crucially, the number of CFBB is linear with respect to the number of control structures. Moreover, the CFBB reduces redundancy and allows to focus only the parts in the control flow that were actually modified in the commit.

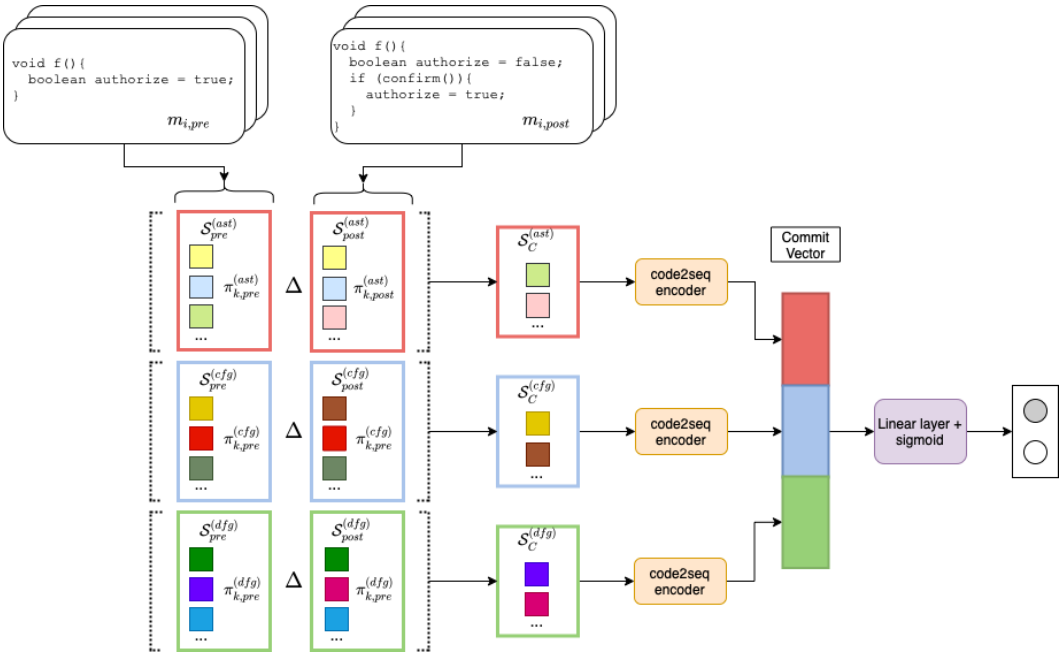

Figure 5: Illustration of our model and how it combines multiple path-based representations of a commit from different graphs.

The model. Figure 5 depicts our proposed model. Instead of using `code2vec`, we rely on the `code2seq` encoder. The reason of this choice is that `code2seq` has been consistently shown to yield better results than `code2vec` (Alon et al., 2019a; Qureshi et al., 2021). Moreover, the use of RNNs to encode the paths makes `code2seq` more adapted to encode longer paths extracted from the DFG and CFG (CFBB).

4 Preliminary Evaluation

As a preliminary evaluation, we test our model on the task of binary classification of security-relevant commits. We use the dataset of Ponta et al. (2019), that contains 1821 commits from 210 public Java projects, of which 921 (50,58%) are labeled *security relevant*, since they contain one or more fixes to existing vulnerabilities, and 900 are labeled *not security relevant.*

The pre-processing pipeline is the following:

1. We download the changed Java files for each commit.
2. Using git we identify the methods that have been modified.
3. We use *Fraunhofer CPG* (`https://github.com/Fraunhofer-AISEC/cpg`) to generate the Code Property Graphs (CPG) (Yamaguchi et al., 2014) of each modified method (both pre- and post-commit versions). CPGs are multi-edged graphs containing all the nodes and edges of the AST, the CFG and the DFG in a unified structure.
4. We extract the path-contexts from the graphs, and we compute the symmetric difference for each method. The path-contexts serve as input to the model.

Because we could not access some of the repositories in the `commit2vec` dataset, the final dataset used for the evaluation contains 1678 commits of which 865 security relevant (51.55%).

On average we found 3708 modified AST path-contexts per commit, compared to an average of 15.81 for the CFBB and 29.37 for DFG paths. Consistently with the original `commit2vec` work, we limit the maximum number of AST path-contexts to 500. If the number of AST path-contexts exceeds this threshold, we randomly sample 500.

	Precision (%)	Recall (%)	F1-score	Accuracy (%)
commit2vec	$61.22 \pm 4.75\%$	$66.13 \pm 5.76\%$	$63.49 \pm 4.64\%$	$61.57 \pm 3.91\%$
ours (AST only)	$67.17 \pm 2.18\%$	$73.51 \pm 4.89\%$	$70.13 \pm 2.77\%$	$67.69 \pm 2.31\%$
ours (AST + DFG)	$68.53 \pm 3.39\%$	$73.85 \pm 1.51\%$	$71.06 \pm 2.25\%$	$68.1 \pm 2.24\%$
ours (AST + CFG)	$\mathbf{68.91 \pm 3.14\%}$	$\mathbf{75.98 \pm 3.43\%}$	$\mathbf{72.20 \pm 2.09\%}$	$\mathbf{69.63 \pm 1.77\%}$
ours (CFG + CFG + DFG)	$68.28 \pm 2.93\%$	$75.42 \pm 4.58\%$	$71.56 \pm 2.09\%$	$69.59 \pm 2.03\%$

Table 1: Evaluation metrics of the models on the task of classification of security-relevant commits.

Because of the unbalance in the number of paths of different types, we chose different encoding sizes for the three partial commit vectors (64 for AST paths, 32 for DFG paths and 16 for CFBB). We used *Pytorch Lightning* (`https://www.pytorchlightning.ai`) to implement the model.

We report the performance evaluation metrics in Table 1. Values show the average and standard deviation over 8 experiments in which we changed the seed. Results confirm the `code2seq` encoder yields significantly better results than `code2vec`, even when only the AST paths are considered. Adding one representation to the baseline AST paths leads to improved metrics, with the highest gains coming from the AST + CFG model that uses Control Flow Basic Blocks (13% improvement in average F1 score and accuracy with respect to `commit2vec`). The "full" model using the three representations at once yields slightly worse results in terms of classification performance with respect to the AST + CFG model, but yields the highest AUC-ROC score as shown in Figure 7.

These results suggests that considering multiple path-based representations can potentially lead to a more effective representation of code changes. In particular, despite the fact that on average the CFBB are half as numerous as DFG paths, the AST + CFG model outperforms the AST + DFG model. This would seem to suggest that the information contained in CFBBs is indeed significant for classification purposes, and that the CFBBs are a sensible approach to represent changes in the control flow of a program.

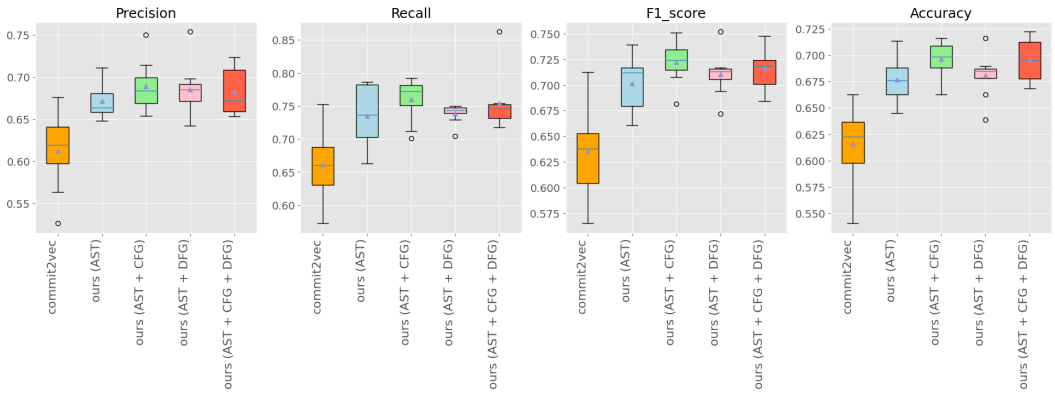

Figure 6: Boxplots of different classification metrics.

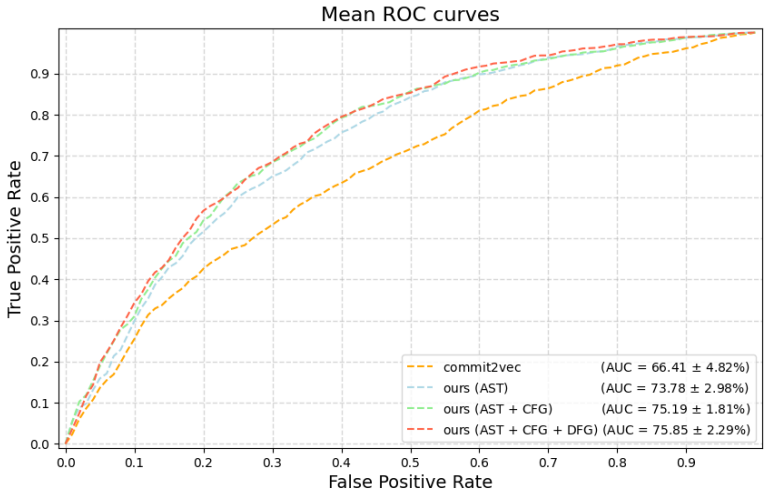

Figure 7: Comparison of the ROC curves of the `commit2vec` model and ours.

5 Conclusion

We introduced a new language-agnostic, path-based method that leverages multiple abstract representations of source code to learn embeddings of code edits. In particular, we defined a new, scalable strategy to extract meaningful paths from Control Flow Graphs, that also reduces redundant information. Our preliminary results corroborate the intuition that relying on multiple code representations is a sensible approach, but we need to extend the experimental campaign on larger and more realistic datasets to fully appreciate the impact of our proposed model. Moreover, we will improve the simple model we used in this work, by considering a hierarchical attention mechanism, as well as studying a Bayesian treatment of the model, by casting every input source (the path-contexts coming from the different graphs) as experts (Hinton, 2002) such that their "uncertainty" can be taken into account when merging their contributions into a single distributed representation.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019a.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019b. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. *SN Computer Science*, 2(3), Mar 2021. ISSN 2661-8907. doi: 10.1007/s42979-021-00566-z. URL http://dx.doi.org/10.1007/s42979-021-00566-z.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun 2020. doi: 10.1145/3377811.3380361. URL http://dx.doi.org/10.1145/3377811.3380361.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020.

Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. Graphcode2vec: Generic code embedding via lexical and program dependence analyses, 2022.

Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software, 2019.

Syed Arbaaz Qureshi, Sonu Mehta, Ranjita Bhagwan, and Rahul Kumar. Assessing the effectiveness of syntactic structure to learn code edit representations, 2021.

Dheeraj Vagavolu, Karthik Chandra Swarna, and Sridhar Chimalakonda. A mocktail of source code representations, 2021.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014. doi: 10.1109/SP.2014.44.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits, 2019.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context, 2021.