Machine Learning for Programming Language Processing

Uri Alon

Machine Learning for Programming Language Processing

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Uri Alon

Submitted to the Senate
of the Technion — Israel Institute of Technology
Av 5781 Haifa June 2021

This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

This thesis is based on the following main publications:

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=i80OPhOCVH2`.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192412. URL `http://doi.acm.org/10.1145/3192366.3192412`.

Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL `https://openreview.net/forum?id=H1gKYo09tX`.

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`.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.

The following publications were part of my PhD research and present results that are supplemental to this work:

Uri Alon, Golan Pundak, and Tara N Sainath. Contextual speech recognition with difficult negative training examples. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6440–6444. IEEE, 2019.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Acknowledgements

First of all, I would like to thank my wonderful advisor, Prof. Eran Yahav. This thesis would not have been possible without his inspirational guidance and his visionary optimism. It has been a real privilege to learn from him how to research.

I am also grateful to Dr. Omer Levy. Although being 10 time zones apart during our joint work, his help was vital to this dissertation and to my development as a researcher.

I would like to thank my co-authors: Meital Zilberstein, Shaked Brody, Roy Sadaka, Yaniv David, Noam Yefet, Elad Nahmias, Chaim Baskin and Ben Finkelshtein; my

office-mates over the years Gail Weiss and Dana Drachsler-Cohen; and Guy Waldman who created the demo websites, which helped me demonstrate and publicize my work.

I would also like to thank Golan Pundak and Tara N. Sainath, who hosted me for an internship at Google New York in the summer of 2018. This internship, although not directly related to the main line of my thesis, had a significant contribution to the development of this dissertation.

I would like to thank my mother-in-law, Anath, for her constant help and support, and my father-in-law, Dudi (Prof. David Engelberg), for the out-of-scope academic advice, and for many pointers along the way.

I am grateful to my parents, Dorit and Nir, who have always encouraged me to excel and follow my dreams.

Finally, I would like to thank my amazing wife, Lee, who has convinced me to pursue graduate studies, supported my every decision, and with our dearest one, Gur, tolerated my deadlines and rejections.

I gratefully acknowledge the generous financial help of the Technion, the Henry and Marilyn Taub Faculty of Computer Science, and the Irwin and Joan Jacobs scholarship.

Contents

List of Figures

Abstract **1**

1 Introduction **3**
1.1 Background . . . 5
1.1.1 AST paths . . . 6
1.2 Overview . . . 7
1.2.1 Contributions . . . 10

2 A General Path-Based Representation for Predicting Program Properties **11**
2.1 Introduction . . . 11
2.2 Overview . . . 15
2.3 AST Paths Representation . . . 16
2.3.1 AST Paths . . . 17
2.3.2 Limiting the Number of Paths . . . 19
2.4 Evaluation . . . 20
2.4.1 Prototype Implementation . . . 21
2.4.2 Experimental Setting . . . 22
2.4.3 Quantitative Evaluation . . . 23
2.4.4 Qualitative Evaluation . . . 28
2.4.5 Impact of Parameter Values . . . 29
2.4.6 Abstractions of AST Paths . . . 30

3 code2vec: Learning Distributed Representations of Code **33**
3.1 Introduction . . . 33
3.1.1 Applications . . . 35
3.1.2 Challenges: Representation and Attention . . . 36
3.1.3 Existing Techniques . . . 37
3.1.4 Contributions . . . 38
3.2 Overview . . . 38
3.2.1 Motivating Example . . . 39

3.3 Background - Representing Code using AST Paths . . . 42
3.4 Model . . . 43
3.4.1 Code as a Bag of Path-Contexts . . . 44
3.4.2 Path-Attention Model . . . 44
3.4.3 Training . . . 47
3.4.4 Using the Trained Network . . . 48
3.4.5 Design Decisions . . . 48
3.5 Evaluation . . . 51
3.5.1 Quantitative Evaluation . . . 52
3.5.2 Evaluation of Alternative Designs . . . 55
3.5.3 Data Ablation Study . . . 56
3.5.4 Qualitative Evaluation . . . 58
3.6 Limitations of the code2vec model . . . 61

4 code2seq: Generating Sequences from Structured Representations of Code 65
4.1 Introduction . . . 65
4.2 Representing Code as AST Paths . . . 66
4.3 Model Architecture . . . 68
4.3.1 Encoder-Decoder Framework . . . 68
4.3.2 AST Encoder . . . 69
4.4 Experiments . . . 70
4.4.1 Code Summarization . . . 71
4.4.2 Code Captioning . . . 74
4.4.3 Code Documentation . . . 75
4.5 Ablation Study . . . 76

5 Structural Language Models of Code 79
5.1 Introduction . . . 79
5.2 Code Generation as Structural Language Modeling . . . 80
5.3 Model Architecture . . . 83
5.3.1 Encoding AST Paths . . . 84
5.3.2 Aggregating Multiple Paths . . . 84
5.3.3 Predicting with a Syntactic Copy Mechanism . . . 85
5.4 Experimental Setup . . . 86
5.4.1 Benchmarks . . . 86
5.4.2 Baselines . . . 87
5.4.3 Implementation and Hyperparameter Settings . . . 88
5.5 Results . . . 88
5.6 Ablation Study . . . 89
5.7 Qualitative Analysis . . . 90

5.8 Related Work 92

6 Limitations of Graph Neural Networks 95

6.1 The Bottleneck of Graph Neural Networks and the Over-squashing Phenomenon 95

6.1.1 Preliminaries 97

6.1.2 The GNN Bottleneck 98

6.2 How Attentive are Graph Attention Networks? 100

7 Additional Applications 105

7.1 Neural Edit Completion 106

7.1.1 Motivating Example 111

7.2 Adversarial Examples for Models of Code 113

7.2.1 Goal 114

7.2.2 Our Approach 115

7.2.3 Motivating Examples 116

7.2.4 Discrete Adversarial Manipulation of Programs (DAMP) 118

8 Conclusion 121

Hebrew Abstract i

List of Figures

1.1 An illustration demonstration of some of the models presented in this thesis. 4

1.2 An abstract illustration of the tradeoff between the learning effort (the effort that is put on the learning model) and the analysis effort (that is made before learning). Learning from the surface text is straightforward and allows employing existing textual approaches, but puts much effort on the learning model. Manually designed features and semantic analysis reduce the learning effort, but make the model language- and task-specific. 6

1.3 A screenshot of the CODE2VEC website: `http://code2vec.org`. A user-provided code snippet is fed into the model, which predicts an appropriate label (a method name of `sort`, in this case). The website also shows the paths that were given the highest attention in the prediction. 7

1.4 A screenshot of the CODE2SEQ website: `http://code2seq.org`. A user-provided code snippet is fed into the model, which predicts an appropriate natural language sequence: "save bitmap to file". The tree on the right illustrates the paths that were given the highest attention while predicting each of the output words. 8

1.5 A screenshot of the AnyCodeGen website: `http://AnyCodeGen.org`. A *partial* user-provided code snippet is fed into the model, with one or more "holes", marked by "??". The model generates a completion for every "hole", such as `stats[i].getPath()`. The website also shows the predicted partial AST of each of the suggested completions. 9

2.1 A JavaScript program and its AST, along with an example of one of the paths. 12

2.2 An overview of our approach. We start with a code snippet $\mathcal{C}$, and extract its path representation to be used as an input to machine learning models. The AST and paths were extracted from the example program in Figure 2.1a. 14

2.3 A JavaScript statement and its partial AST. 18

2.4 An example statement and its AST, with an example of a path between the *SymbolVar* terminals that represent `a` and `d`. The length of this path is 4, and its width is 3. 19
2.5 An example of a typical program where the maximal path length is relatively small, but the width can be large. 20
2.6 Example of a Python program with stripped names and with predictions produced using our AST paths. 27
2.7 Example of a JavaScript program with stripped names, with predictions produced using our AST paths and an online version of UnuglifyJS at nice2predict.org. This is the default example shown at nice2predict.org. . . 28
2.8 Examples of Java programs with stripped names and with predictions produced using our AST paths. We deliberately selected challenging examples in which the prediction cannot be aided by specific classes and interfaces. 28
2.9 Accuracy results of AST paths with CRFs, for the task of variable naming in JavaScript, for different combination values of *max_length* and *max_width* (UnuglifyJS is presented here for comparison). 30
2.10 The accuracy of each abstraction method compared to the consumed training time, for the task of variable naming in Java 31

3.1 A code snippet and its predicted labels as computed by our model. . . . 34
3.2 Examples of three methods that can be easily distinguished by our model despite having similar syntactic structure: our model successfully captures the subtle differences between them and predicts meaningful names. Each method portrays the top-4 paths that were given the most attention by the model. The widths of the colored paths are proportional to the attention that each path was given. 39
3.3 The top-4 attended paths of Figure 3.2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (**red** ①: **0.23**, **blue** ②: **0.14**, **green** ③: **0.09**, **orange** ④: **0.07**). 40
3.4 The architecture of our path-attention network. A *fully connected layer* learns to combine embeddings of each path-context with itself; attention weights are learned using the combined context vectors and used to compute a *code vector*. The code vector is used to predict the label. . . 45
3.5 Our model achieves significantly better results than the baselines and in shorter time. 54
3.6 Example predictions from our model, with the top-4 paths that were given the most attention for each code snippet. The width of each path is proportional to the attention it was given by the model. 57

3.7 An example for a method name prediction, portrayed on the AST. The top-four path-contexts were given a similar attention, which is higher than the rest of the path-contexts. 59

3.8 An example for a method name prediction, portrayed on the AST. The width of each path is proportional to the attention it was given. 60

4.1 Example of (a) code summarization of a Java code snippet, and (b) code captioning of a C# code snippet, along with the predictions produced by our models. The highlighted paths in each example are the top-attended paths in each decoding step. Because of space limitations we included only the top-attended path for each decoding step, but hundreds of paths are attended at each step. 66

4.2 An example of two Java methods that have exactly the same functionality. Although these methods have different *sequential* (token-based) representations, repeating paths, which might differ in only a single node (a `ForStmt` node instead of a `Do`-while node), will be revealed if we consider syntactic patterns. 67

4.3 Our model encodes each AST path with its values as a vector, and uses the average of all of the k paths as the decoder's start state. The decoder generates an output sequence while attending over the k encoded paths. 69

4.4 Visualization of the F1 score of our model compared to the baselines, for the code summarization task, across datasets. Our model achieves significantly higher results than the baselines. 73

4.5 F1 score compared to the length of the input code. This experiment was performed for the code summarization task on the Java-med test set. All examples having more than 30 lines were counted as having 30 lines. . . 74

4.6 Visualization of the BLEU score of our model compared to the baselines, for the code captioning task. Our model achieves significantly higher results than the baselines. 75

5.1 Examples from the Java (left) and C# (right) test sets. The highlighted expression in each example is the target p, which our models correctly generated from the rest of the snippet. Additional and larger examples can be found in the supplementary material. 80

5.2 The subtree representing `x > 1` is generated given its surrounding tree. At each step, the model generates the next node (denoted by ?) of $path_1$, $path_2$ and $path_3$ using the root path $\mathcal{R}$. Dashed lines denote the AST structure; solid lines denote AST paths. Most AST paths are omitted from the figure, for clarity. 81

5.3 Augmenting the AST with $\texttt{EOS}_{node}$ and $\texttt{EOS}_{tok}$ nodes. 83

5.4 Examples for cases where the top candidate is a "tree-match" (marked with 🌳), but only the second candidate is an "exact match" (marked with ✓ in bold). Predictions that are logically equivalent to the ground truth are marked with ↔. 91

5.5 An example from our test set in which a compiler-guided generation could filter out non-compiling candidates, and thus rank the ground truth *second* instead of *fifth*. Four out of the five candidates are "tree-match" (marked with 🌳), the fifth candidate is an "exact match" (marked with ✓ in bold), and only the second and the fifth candidate compile (marked with ⚙). 92

6.1 The bottleneck that existed in RNN seq2seq models (before attention) is strictly more harmful in GNNs: information from a node's exponentially-growing receptive field is compressed into a fixed-size vector. Black arrows are graph edges; red curved arrows illustrate information flow. . . . 96

6.2 The NeighborsMatch problem: green nodes have blue neighbors and an alphabetical label. The goal is to predict the label (A, B, or C) of the green node that has the same number of blue neighbors as the target node in the same graph. In this example, the correct label is **C**, because the target node has *two* blue neighbors, like the node marked with C in the same graph. 99

6.3 Standard GAT (Figure 6.3a) computes *static* attention: the ranking of attention coefficients is global for all nodes in the graph, and is unconditioned on the query node. For example, all queries ($q0$ to $q9$) attend mostly to the 8th key ($k8$). In contrast, GATv2 (Figure 6.3b) can actually compute *dynamic* attention, where every query has a different ranking of attention coefficients of the keys. 101

7.1 Examples of EditCompletion. The input consists of a program fragment $\mathcal{P}$ and edits that occurred in the context that transformed $\mathcal{C}$ into $\mathcal{C}'$. The output is $\Delta_{\mathcal{P}}$ – an edit script that describes the likely edit. Applying $\Delta_{\mathcal{P}}$ to $\mathcal{P}$ results in $\mathcal{P}'$ – the code after the edit. 107

7.2 An example of two edits. These examples are different and the edits operate on different values. However, observing the structure of these edits reveals the similarity between them and allows a learning model to generalize better. This similarity is expressed as almost identical AST paths. For simplicity, only the program fragment that should be edited $\mathcal{P}$ is shown, without the context $\mathcal{C}$. 108

7.3 An EditCompletion example from our test set. Figure 7.3a shows the edit that transforms $\mathcal{C}$ into $\mathcal{C}'$ – overloading the function `AddNavigation`. Figure 7.3e shows $\mathcal{P}$ and $\mathcal{P}'$ as code in red and green, respectively. Figure 7.3b depicts the partial AST and the first three edit operations of the edit. Figure 7.3c shows the AST after applying the first three operations, and shows the next three operations as AST paths. Figure 7.3d illustrates the AST after performing all operations, resulting in an AST that corresponds to $\mathcal{P}'$. Every edit operation is represented by an AST path having the same color and number as the edit command. Dotted contours represent subtrees that will be affected by applying these operations. 111

7.4 A Java snippet `f1` is classified correctly as `sort` by the model of `code2vec.org`. Given `f1` and the target `contains`, our approach generates `f2` by renaming `array` to `ttypes`. Given the target `escape`, our approach generates `f3` by adding an unused variable declaration of `int upperhexdigits`. Additional examples can be found in Yefet et al. (2020). 114

7.5 Perturbing a variable name: the original variable name is represented as a one-hot vector over the variable-name vocabulary. After perturbation, the vector is no longer one-hot. We apply argmax to find the most likely adversarial name, resulting with another one-hot vector over the variable-name vocabulary. 115

7.6 A C# VarMisuse example which is classified correctly as `DestinationType` in the method `Equals` by the GGNN model of Allamanis et al. (2018). Given the code in Figure 7.6a and the target `SourceType`, our approach renames a local variable `destination` in *another method* to the specific name `scsqbhj`, making the model predict the wrong variable in the method `Equals`, thus ("maliciously") introducing a real bug in the method `Equals`. Additional examples are shown in Yefet et al. (2020). . 117

Abstract

Over the past decade, the *deep learning* revolution, driven by artificial neural networks, has transformed a broad range of areas in computer science such as computer vision, speech recognition, and natural language processing (NLP). In parallel, the number of open-source, publicly available codebases has grown dramatically, enabling the application of neural networks to a wide range of programming-related tasks, a field that we dub *Programming Language Processing* (PLP).

Yet, the problem of *representing* programs in machine- and deep-learning models remained an open question. Obviously, programs do not have a straightforward tensorial representation as images do. Although a program *can* be represented as a sequence of tokens like a natural language text, programs are far more *structured* than free-text, since programs must comply with a rigid and rich syntax, defined by a context-free grammar. Furthermore, every programming language has predefined *semantics* that describe what syntactically valid programs mean and do.

This dissertation focuses on this general problem: representing programs in machine- and deep-learning models, in ways that facilitate learning while capturing as much information as possible, and keeping the model as general as possible. This thesis introduces the *AST paths* approach, which represents programs using paths from the program's Abstract Syntax Tree (AST). The AST paths representation allows building powerful and accurate neural models, while keeping them lightweight and scalable. Specifically, this thesis shows how these models can be trained on millions of examples, in tasks that include predicting properties of individual program elements, predicting properties of code snippets, generating a natural language sequence from a given source code snippet, and generating code completions. These models were publicly released as online interactive demos along with open-source implementations and datasets. Some of the models, such as CODE2VEC and CODE2SEQ, are highly popular and widely used in academia and industry.

Finally, this dissertation studies theoretical differences between different program representations. These studies revealed broader foundational limitations of another popular representation, the graph neural network framework.

Chapter 1

Introduction

The past decade has seen monumental advances in machine learning. Specifically, deep learning, powered by artificial neural networks, has transformed a broad range of areas in computer science such as computer vision (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012), and natural language processing (NLP) (Collobert et al., 2011), and has spread to a variety of other scientific disciplines such as biology (Angermueller et al., 2016), chemistry (Gilmer et al., 2017), and physical simulation (Sanchez-Gonzalez et al., 2020). In parallel, despite great popularity of rule-based (i.e., deterministic, non-learning) coding assistance tools, the vast majority of programming, fixing, naming, and debugging efforts are still performed by human programmers. Programming requires expertise and is time-consuming; further, even expert programmers lookup for online help frequently, and still create bugs. Thus, the dramatically growing availability of open-source code repositories creates exciting new opportunities for employing deep learning for a wide range of programming-related applications, a field that we dub *Programming Language Processing* (PLP).

Nonetheless, before even considering the actual learning algorithm or model, *representing* programs in machine- and deep-learning models remained an open problem. That is, it is unclear which facets of programs capture the most information about the program, while remaining compact and generalizable. Further, it is unclear how to input these facets into a learning algorithm. While an image can be represented as a matrix or a tensor of pixels, and a natural language sentence can be represented as a one-dimensional sequence of words, it is unclear whether programs can be described using such simple input representations. Although a program *can* be represented as a sequence of tokens like natural language text, programs are far more *structured* than free-text, since a (valid) program must comply with a rigid and rich syntax, defined by a context-free grammar. Thus, a program may only *look* like text, where in fact, it is more of a *tree* than a sequence. Further, every programming language has predefined *semantics* that describe what syntactically valid programs mean and do. So, providing the learning model with some information about the semantics of the language, along with the program itself, might help the model in performing some tasks. However,

excessive dependence on semantics restricts the *scalability* of the learning models, limits the feasible amount of training data, and hurts the practical applicability of such models.

Thus, the main question investigated in this thesis is:

> *How can we represent programs in learning models*
> *to achieve accurate, computationally effective, and large-scale models?*

Figure 1.1 shows an illustrative demonstration of some of the models presented in this thesis.

Research impact In this thesis, we present a line of work that began as a contribution in the field of programming languages, but quickly drew attention and impacted a variety of other fields such as software engineering (Henkel et al., 2018; Cambronero et al., 2019; Liu et al., 2019b; Kang et al., 2019), machine learning (Zügner et al., 2021; Yao et al., 2021; Fernandes et al., 2019; Liu et al., 2021), NLP (Feng et al., 2020; Fujita et al., 2020; Panthaplackel et al., 2020; Yu et al., 2020) and security (Schuster et al., 2021; Sonnekalb, 2019; Compton et al., 2020; Lacomis et al., 2019). Further, our intuitions from structural representations of code have led to identifying general insights that drew attention in the general geometric deep learning community (Valsesia et al., 2021; Bronstein et al., 2021; Morris et al., 2021; Kurin et al., 2021; Lukovnikov and Fischer, 2021; Kreuzer et al., 2021; Godwin et al., 2021). Our code and online demos are used by thousands of people and draw positive feedback and enthusiastic response.

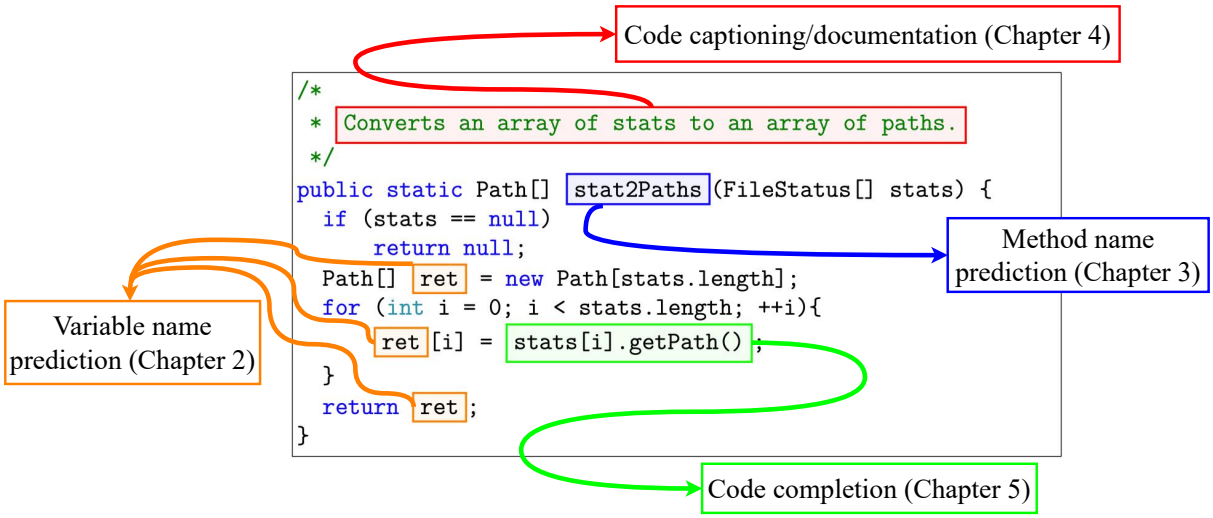


Figure 1.1: An illustration demonstration of some of the models presented in this thesis.

1.1 Background

Representing programs in machine- and deep-learning models has always been an open problem. Here we discuss the main existing approaches.

Sequential representation of code A direct approach is to frame the representation problem of code as text, or a sequence of tokens. Several works have taken this approach and modeled code as a sequence. These sequences could be of tokens (Allamanis et al., 2014; Allamanis and Sutton, 2013; Allamanis et al., 2016; Movshovitz-Attias and Cohen, 2013; White et al., 2015; Hindle et al., 2012; Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017; Pradel and Sen, 2018; Vasic et al., 2019; Kanade et al., 2020), and also characters (Bielik et al., 2017) and API calls (Raychev et al., 2014). This straightforward approach allows employing off-the-shelf approaches and architectures from NLP. While sometimes obtaining satisfying results, these models treat code as a sequence rather than a tree. This necessitates implicit re-learning of the (predefined) syntax of the programming language, often *requiring* large models, large amounts of data, and very long training times.

Semantic representations of code On the other extreme, *semantic* representations of code are appealing because they allow providing the learning model with all the knowledge that is already "known", preserving the model's learning capacity. Such semantic representations usually represent the code as a (neural or non-neural) graph. Nodes are program elements, and edges represent syntactic and semantic relations between the elements (Allamanis et al., 2018; Brockschmidt et al., 2019; Hellendoorn et al., 2019; Guo et al., 2021; Peng et al., 2021) or other manually designed features (Raychev et al., 2015; Dinella et al., 2020; Zügner et al., 2021). Although this semantic information can significantly ease learning, it also adds dependence on this information, which is not always given. For example, Brockschmidt et al. (2019) compile all training code projects before training, to provide the learning phase with information about types. This severely limits the model to perform inference only for complete projects that actually compile: if a project from the test-set does not compile, or we wish to perform inference on a partial code snippet (without its surrounding code, files, and dependencies) – the ability of the model to provide useful predictions is acutely hurt. Further, depending on semantics makes the model language- and task-specific: extending such a model to a *different prediction task* requires rethinking of the needed semantic relations and features. Extending such a model to a *different programming language* requires re-designing the semantic analysis, which is not always possible due to the nature of different languages. For example, extending a semantic model that was designed for Java to a language such as Python may not be able to use the same amount of semantic information, due to the static typing of Java and the dynamic typing of Python.

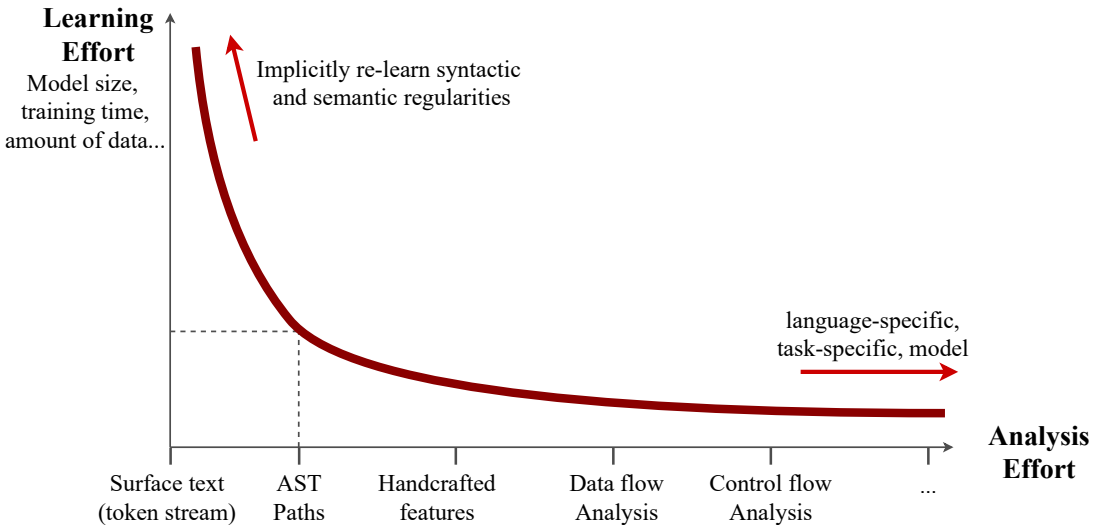


Figure 1.2: An abstract illustration of the tradeoff between the learning effort (the effort that is put on the learning model) and the analysis effort (that is made before learning). Learning from the surface text is straightforward and allows employing existing textual approaches, but puts much effort on the learning model. Manually designed features and semantic analysis reduce the learning effort, but make the model language- and task-specific.

1.1.1 AST paths

A useful and interesting sweet-spot between the sequential and the semantic representations, which is one of the main topics of this dissertation, is AST paths representation. AST paths represent programs as paths between nodes in the program's Abstract Syntax Tree (AST). These paths can be thought of as "structural n-grams", which make this approach general and simple. The use of syntax reduces much of the learning effort from AST paths-based models compared to sequential models, because the model does not need to waste capacity by learning that "an opening parenthesis must be followed by a closing parenthesis", for example. Compared to semantic models, AST paths are much more general, because the same model and approach can be used for different tasks and languages, while only replacing the parser.

Figure 1.2 summarizes the tradeoff discussed above between the learning effort and the generalization limitation. In the following chapters, we show the applicability and generality of the AST paths approach. We note that there are other works that follow the purely-syntactic approach (Allamanis and Sutton, 2014; Bielik et al., 2016; Yin and Neubig, 2017; Rabinovich et al., 2017; Yin et al., 2019; Yin and Neubig, 2018; Yao et al., 2021; Kim et al., 2021). Some of these will be discussed where relevant; others use GNN to encode the AST, and are thus susceptible to the GNN bottleneck that we present in Chapter 6.

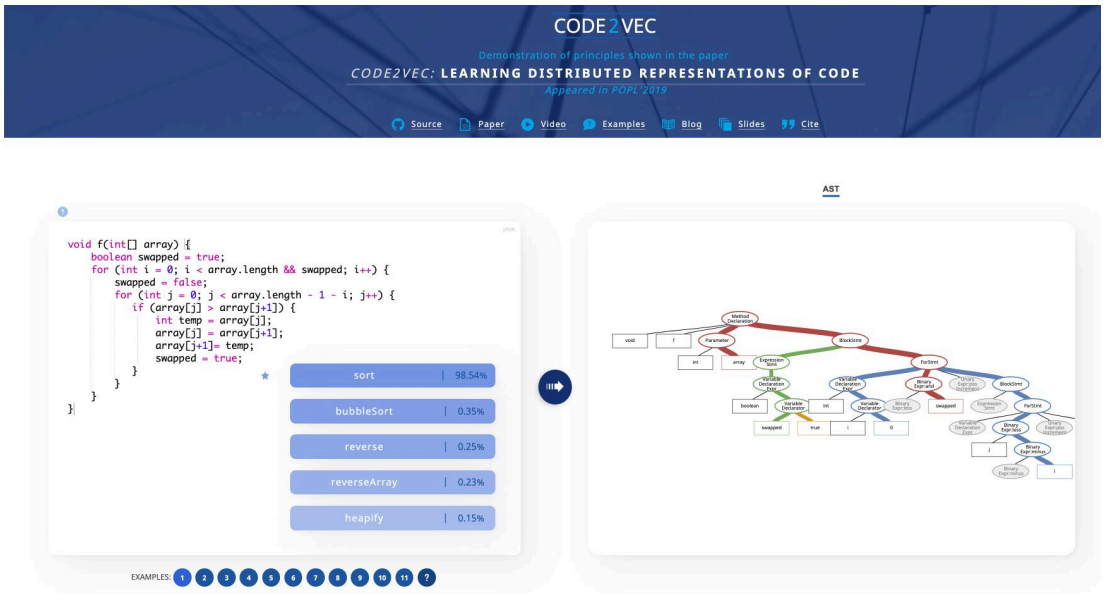


Figure 1.3: A screenshot of the CODE2VEC website: `http://code2vec.org`. A user-provided code snippet is fed into the model, which predicts an appropriate label (a method name of `sort`, in this case). The website also shows the paths that were given the highest attention in the prediction.

1.2 Overview

Next, we give a brief overview of some of the main methods presented in this thesis. The exact details and formal definitions are provided in the respective chapters.

code2vec In Chapter 3, we present a neural model for representing snippets of code as continuous distributed vectors ("code embeddings"). The main idea is to represent a code snippet as a single fixed-length *code vector*, which can be used to predict semantic properties of the snippet. To this end, code is first decomposed to a collection of paths in its AST. Then, the network learns the atomic representation of each path while *simultaneously* learning how to aggregate a set of them.

We demonstrate the effectiveness of our approach by using it to predict a method's name from the vector representation of its body. We evaluate our approach by training a model on a dataset of 12M methods. We show that code vectors trained on this dataset can predict method names from files that were unobserved during training. Furthermore, we show that our model learns useful method name vectors that capture semantic similarities, combinations, and analogies. Figure 1.3 shows a screenshot of our online demo (`http://code2vec.org`).

code2seq The ability to *generate natural language sequences from source code snippets* has a variety of applications such as code summarization, documentation, and retrieval. Sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), have achieved state-of-the-art performance on these tasks by treating

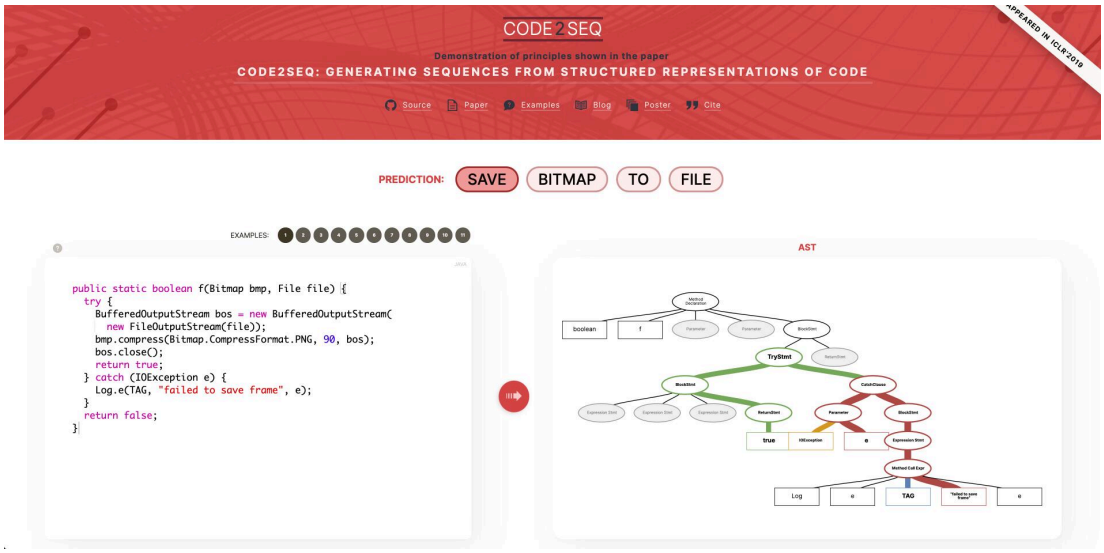


Figure 1.4: A screenshot of the CODE2SEQ website: `http://code2seq.org`. A user-provided code snippet is fed into the model, which predicts an appropriate natural language sequence: "save bitmap to file". The tree on the right illustrates the paths that were given the highest attention while predicting each of the output words.

source code as a sequence of tokens. In Chapter 4, we present CODE2SEQ: an alternative approach that leverages the syntactic structure of programming languages to better encode source code. Our model represents a code snippet as the set of compositional paths in its AST and uses attention to select the relevant paths while decoding.

We demonstrate the effectiveness of our approach for two tasks, two programming languages, and four datasets of up to 16M examples. Our model significantly outperforms previous models that were specifically designed for programming languages, as well as state-of-the-art NMT models. Figure 1.4 shows a screenshot of our online demo (`http://code2seq.org`).

SLM In Chapter 5, we address the problem of *any-code completion* – generating a missing piece of source code in a given program without any restriction on the vocabulary or structure. We introduce a new approach to any-code completion that leverages the strict syntax of programming languages to model a code snippet as a tree – *structural language modeling* (SLM). SLM estimates the probability of the program's abstract syntax tree (AST) by decomposing it into a product of conditional probabilities over its nodes. We present a neural model that computes these conditional probabilities by considering all AST paths leading to a target node. Unlike previous techniques that have severely restricted the kinds of expressions that can be generated in this task, our approach can generate arbitrary code in any programming language. Our model significantly outperforms both seq2seq and a variety of structured approaches in generating Java and C# code. Figure 1.5 shows a screenshot of our online demo (`http://AnyCodeGen.org`).

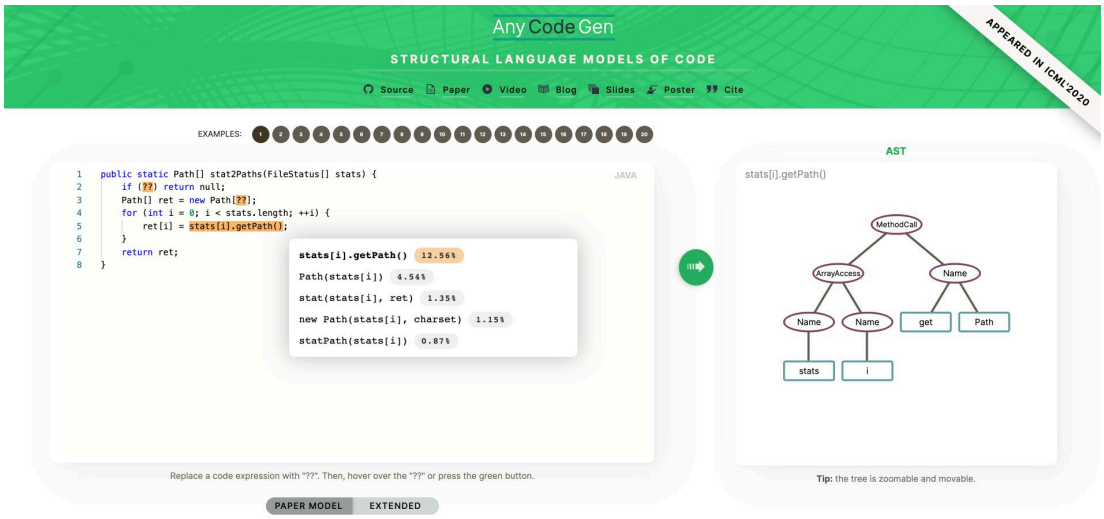


Figure 1.5: A screenshot of the AnyCodeGen website: `http://AnyCodeGen.org`. A *partial* user-provided code snippet is fed into the model, with one or more "holes", marked by "??". The model generates a completion for every "hole", such as `stats[i].getPath()`. The website also shows the predicted partial AST of each of the suggested completions.

The bottleneck of graph neural networks While studying different representations of code, we noticed that graph neural networks, although very general and versatile, fail to learn long-range patterns in the training data. When trained on programming tasks that *depend* on long-range interactions, we found that GNNs usually *overfit* on short-range artifacts in the data. This phenomenon was surprising, because AST paths had no problem learning long-range signals. Searching through the literature, it turned out that since the proposal of the GNNs (Gori et al., 2005; Scarselli et al., 2008), their struggle to propagate information between distant nodes in the graph was one of the major problems in training them. In Section 6.1 we propose a new explanation for this problem: GNNs are susceptible to a *bottleneck* when aggregating messages across a long path. This bottleneck causes the *over-squashing* of exponentially growing information into fixed-size vectors. As a result, GNNs fail to propagate messages originating from distant nodes and perform poorly when the prediction task depends on long-range interaction. In Section 6.1, we highlight the inherent problem of over-squashing in GNNs: we demonstrate that the bottleneck hinders popular GNNs from fitting long-range signals in the training data; we further show that GNNs that absorb incoming edges equally, such as GCN and GIN, are *more susceptible* to over-squashing than GAT and GGNN; finally, we show that prior work, which extensively tuned GNN models of long-range problems, suffer from over-squashing, and that breaking the bottleneck improves their state-of-the-art results without any tuning or additional weights.

Aspect	Instantiation in this thesis	
Code representation	AST paths	Chapter 2
	code2vec	Chapter 3
	Structural language models	Chapter 5
Application	CODE2SEQ	Chapter 4
	Adversarial examples for models of code	Section 7.2
	Edit completion	Section 7.1
	Reverse engineering using neural networks	David et al. (2020)
Foundations and theory	The GNN bottleneck	Section 6.1
	Expressiveness of graph attention networks	Section 6.2

Table 1.1: Aspects of programming language processing we discuss in this thesis.

1.2.1 Contributions

To summarize, the main contributions of this thesis are:

- A new general and language-agnostic family of representations for program elements in machine learning models: AST paths (Chapter 2).
- An approach for embedding an *entire code snippet* as a vector, based on its bag of AST paths. We demonstrate this approach in a large-scale neural model that predicts a label from a code vector (Chapter 3).
- A neural approach to "translate" source code into a natural language sequence. We implement this approach in neural models for code summarization, code captioning, and code documentation (Chapter 4).
- A novel approach to *language modeling of code*: structural language modeling (SLM). We demonstrate this approach in a neural model that estimates the probability of code snippets, and can perform any-code completion by searching for the most probable completion in a given context (Chapter 5).
- Recognition of inherent limitations of graph neural networks (GNNs) in modeling code and other kinds of structural data: we introduce the *over-squashing* phenomenon whose implications were observed before, but their source was never explained (Section 6.1); additionally, we identify that the popular Graph *Attention* Network, does not, in fact, compute the standard and powerful form of attention (Section 6.2).
- Additional aspects and applications of PLP: the vulnerability of models of code to semantic-preserving *adversarial examples* (Section 7.2), and a structural model for representing and predicting *code edits* (Section 7.1).

Table 1.1 summarizes the different aspects of programming language processing that we consider in this thesis.

Chapter 2

A General Path-Based Representation for Predicting Program Properties

Predicting program properties such as names or expression types has a wide range of applications. It can ease the task of programming, and increase programmer productivity. A major challenge when learning from programs is *how to represent programs in a way that facilitates effective learning.*

We present a *general path-based representation* for learning from programs. Our representation is purely syntactic and extracted automatically. The main idea is to represent a program using paths in its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of code rather than treating it as a flat sequence of tokens.

We show that this representation is general and can: cover different prediction tasks, drive different learning algorithms (for both generative and discriminative models), and work across different programming languages.

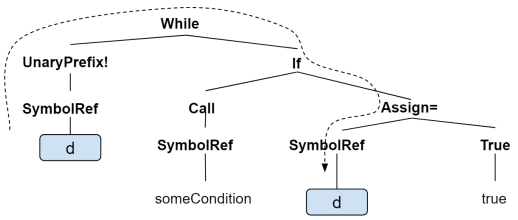
We evaluate our approach on the tasks of predicting variable names, method names, and full types. We use our representation to drive both CRF-based and word2vec-based learning, for programs of four languages: JavaScript, Java, Python and C#. Our evaluation shows that our approach obtains better results than task-specific handcrafted representations across different tasks and programming languages.

2.1 Introduction

Leveraging machine learning models for predicting program properties such as variable names, method names, and expression types is a topic of much recent interest (Raychev et al., 2015; Allamanis et al., 2015a, 2016; Raychev et al., 2016a; Bielik et al., 2016; Maddison and Tarlow, 2014). These techniques are based on learning a statistical model from a large amount of code and using the model to make predictions in new programs.

```
while (!d) {
    if (someCondition()) {
        d = true;
    }
}
```

(a) A simple JavaScript program.



(b) The program's AST, and example of an AST path.

Figure 2.1: A JavaScript program and its AST, along with an example of one of the paths.

A major challenge in these techniques (and in many other machine-learning problems) is how to represent instances of the input space to facilitate learning (Shalev-Shwartz and Ben-David, 2014). Designing a program representation that enables effective learning is a critical task that is *often done manually for each task and programming language*.

Our approach We present a novel program representation for learning from programs. Our approach uses different path-based abstractions of the program's abstract syntax tree. This family of path-based representations is natural, general, fully automatic, and works well across different tasks and programming languages.

AST paths We define AST paths as paths between nodes in a program's abstract syntax tree (AST). To automatically generate paths, we first parse the program to produce an AST, and then extract paths between nodes in the tree. We represent a path in the AST as a sequence of nodes connected by up and down movements, and represent a program element as the set of paths that its occurrences participate in. Figure 2.1a shows an example JavaScript program. Figure 2.1b shows its AST, and one of the extracted paths. The path from the first occurrence of the variable `d` to its second occurrence can be represented as:

$$\text{SymbolRef} \uparrow \text{UnaryPrefix!} \uparrow \text{While} \downarrow \text{If} \downarrow \text{Assign=} \downarrow \text{SymbolRef}$$

This is an example of a pairwise path between leaves in the AST, but in general the family of path-based representations contains n-wise paths, which do not necessarily span between leaves and do not necessarily contain all the nodes in between. Specifically, we consider several choices of subsets of this family in Section 2.3.

Using a path-based representation has several major advantages over existing methods:

1. Paths are generated automatically: there is no need for manual design of features aiming to capture potentially interesting relationships between program elements. This approach extracts unexpectedly useful paths, without the need for an expert

to design features. The user is required only to choose a subset of our proposed family of path-based representations.

2. This representation is useful for any programming language, without the need to identify common patterns and nuances in each language.

3. The same representation is useful for a variety of prediction tasks, by using it with off-the-shelf learning algorithms or by simply replacing the representation of program elements in existing models (as we show in Section 2.4.3).

4. AST paths are purely syntactic, and do not require any semantic analysis.

Tasks In this work, we demonstrate the power and generality of AST paths on the following tasks:

- **Predicting names for program elements** Descriptive and informative names for program elements such as variables and classes play a significant role in the readability and comprehensibility of code. Empirical studies have shown that choosing appropriate names makes code more understandable (Takang et al., 1996), reduces code maintenance efforts, and leads to fewer bugs (Butler et al., 2009). A study in the Psychology of Programming suggests that the ways in which programmers choose names reflect deep cognitive and linguistic influences (Liblit et al., 2006). A meaningful name describes the role of a program element, carries its semantic meanings, and indicates its usage and behavior throughout the program.

- **Predicting method names** Good method names adequately balance the need to describe the internal implementation of the method and its external usage (Høst and Østvold, 2009). When published in a popular library's API, descriptive and intuitive method names facilitate the use of methods and classes, while poorly chosen names can doom a project to irrelevance (Allamanis et al., 2015a). Although method names are clearly program elements and can be predicted by the previous task, in this task we assumes that all the other names in the method are given, along with the names of the elements around the method invocation, when available in the same file.

- **Predicting expression types** Statistical type prediction allows (likely) types of expressions to be inferred without the need for type inference, which often requires a global program view (possibly unavailable, e.g., in the case of snippets from sites such as StackOverflow).

Raychev et al. (2015) used relations in the AST as features for learning tasks over programs. They defined an explicit grammar to derive features which capture specific relationships between nodes in the AST of JavaScript programs, as well as relations

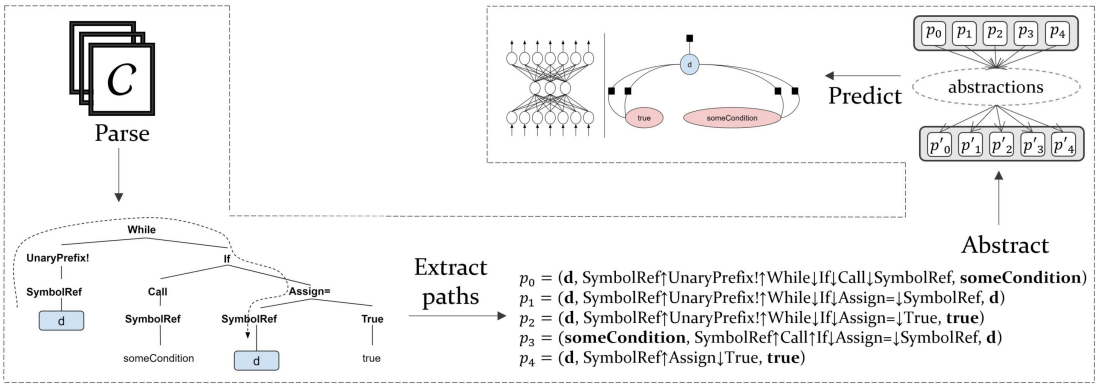


Figure 2.2: An overview of our approach. We start with a code snippet $\mathcal{C}$, and extract its path representation to be used as an input to machine learning models. The AST and paths were extracted from the example program in Figure 2.1a.

produced by language-specific semantic analysis, such as "may call" and "may access". We show that our *automatic general representation* performs better than their features for their original task, and also generalizes to drive two different learning algorithms and three different prediction tasks, over different programming languages.

Paths in an AST have also been used by Bielik et al. (2016) and by Raychev et al. (2016a,b) for a different goal: identifying context nodes. These works do not use the paths themselves as a representation of the input, and the prediction is only affected by the context node that was found on the other end of the path. In our work, we use the path itself as a representation of a program element. Therefore, the prediction depends not only on the context node but also on *the way it is related* to the element in question.

Allamanis et al. (2015a) defined the challenging task of predicting method names, which can be viewed as a form of function summarization (Allamanis et al., 2016). We show that our representation performs better by being able to learn across projects.

Contributions The main contributions of this work are:

- A new, general family of representations for program elements. The main idea is to use AST paths as representations of code.

- A cross-language tool called PIGEON, which is an implementation of our approach for predicting program element names, method names, and types.

- An evaluation on real-world programs. Our experiments show that our approach produces accurate results for different languages (JavaScript, Java, Python, C#), tasks (predicting variable names, method names, types) and learning algorithms (CRFs, word2vec). Furthermore, for JavaScript and Java, where previous methods exist, our automatic approach produces more accurate results.

2.2 Overview

In this section, we illustrate our approach with a simple JavaScript program for the task of predicting names; as we show in later sections, the same approach also applies to other tasks, other languages, and other learning algorithms.

Given a program with non-descriptive names, our goal is to predict likely names for local variables and function parameters. The non-descriptive names could have been given by an inexperienced programmer, or could have been the result of deliberate stripping. In the latter case, we refer to such a program as a program with *stripped names*. Stripping names can be part of a minification process in JavaScript, or obfuscation in Java and other languages.

Consider the code snippet of Figure 2.1a. This simple snippet captures a common programming pattern in many languages. Suppose that we wish to find a better name for the variable `d`.

Program element representation The main idea of our approach is to extract paths from the program's AST and use them to represent an element, such as the variable `d`, in a machine learning model. Figure 2.2 shows an overview of this process. First, we parse the query program to construct an AST. Then, we traverse the tree and extract paths between AST nodes. To simplify presentation, in this example we only consider pairwise paths between AST leaves. We assume that a path is represented as a sequence of AST nodes, linked by up and down movements (denoted by arrows). As we describe in Section 2.3, the path can also connect a leaf and a higher nonterminal in the AST, connect several nodes (n-wise path), and can be abstracted in different levels.

Consider the p_1 in Figure 2.2, between the two occurrences of the variable `d`:

$$\text{SymbolRef} \uparrow \text{UnaryPrefix!} \uparrow \text{While} \downarrow \text{If} \downarrow \text{Assign=} \downarrow \text{SymbolRef} \tag{I}$$

The path expresses the fact that the variable `d` is used, with negation, as a stopping condition of a "while" loop, and then assigned a new value if an "if" condition inside the loop evaluates to `true`. This path alone expresses the fact that `d` is the stopping condition of the loop.

The path p_4 in Figure 2.2, between the variable `d` and the value `true` is:

$$\text{SymbolRef} \uparrow \text{Assign=} \downarrow \text{True} \tag{II}$$

This path captures the fact that the assignment changes the value of `d` to `true`, and therefore it is indeed the assignment that stops the loop.

Prediction By observing these two paths, a programmer is likely to name `d` "done", "complete", "stop" or something similar. Indeed, a learning model that was trained using our representation predicts that the most likely name for the variable is `done`,

and neither “done”, “complete”, nor any similar name was predicted by past work for this example.

Learning algorithms The learning model can vary between different algorithms, presenting tradeoffs of efficiency and accuracy. In Section 2.4.3 we show that both CRFs and word2vec can be used for this prediction task. In both of these learning algorithms, using AST paths produces better results than the alternative representations, whether they are manually designed or sequence-based representations.

Path abstractions Automatic generation may produce a prohibitively large number of paths. To control the number of paths, *higher levels of abstraction* can be used. Instead of representing the whole path node-by-node, it can be further abstracted by keeping only parts of it, which results in similar paths being represented equally, as we show in Section 2.4.6. Another way to control the number of paths is to limit the number of extracted paths. We provide hyper-parameters (i.e., model configurations that are not tuned by the optimization process) that control the maximal length and width of AST paths. The number of extracted paths can be further reduced using downsampling, with minimal impact on accuracy and a significant saving in training time (Section 2.4.3). These methods make the accuracy – training time tradeoff tunable.

Key aspects The example highlights several key aspects of our approach:

- Useful paths such as path I span multiple lines of the program, but are also supported by shorter paths like path II, which only spans a single program line. Short paths alone are not enough to predict a meaningful name. Making a prediction using all paths that an element participates in provides a rich context for predicting the name of the element.
- No special assumptions regarding the AST or the programming language were made, making the same mechanism useful in other languages in a similar way.
- This representation can be plugged into existing models as a richer representation of the input code, without interfering with the learning algorithm itself.
- AST paths can distinguish between programs that previous works could not.
- In addition to predicting `done`, a model trained with AST paths can propose several semantically similar names, as we demonstrate in Section 2.4.3. This shows that AST paths are strong indicators of the program element’s semantics.

2.3 AST Paths Representation

In this section, we formally describe the family of AST paths.

2.3.1 AST Paths

To learn from programs, we are looking for a representation that captures interesting properties of ASTs while keeping it open for generalization. One way to obtain such a representation is to decompose the AST to parts that repeat across programs but can also discriminate between different programs. One such decomposition is into paths between nodes in the AST. We note that in general we consider n-wise paths, i.e., those that have more than two ends, but for simplicity we base the following definitions on pairwise paths between AST terminals.

We start by defining an AST, an AST-path, a path-context and an abstract path-context.

Definition 2.3.1 (Abstract Syntax Tree)**.** An Abstract Syntax Tree (AST) for a code snippet $\mathcal{C}$ is a tuple $\langle N, T, X, s, \delta, val \rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of terminal values, $s \in N$ is the root node, $\delta : N \to (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children, and $val : T \to X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.

For convenience, we also define $\pi : (N \cup T) \to N$, the inverse function for δ. Given a node, this function returns its parent node, such that for every two terminal or nonterminal nodes $y_1, y_2 \in (N \cup T)$, one is the parent node of the other if and only if the latter is in the list of children of the former: $\pi(y_1) = y_2 \iff y_1 \in \delta(y_2)$. In the case of the start symbol, its parent is undefined.

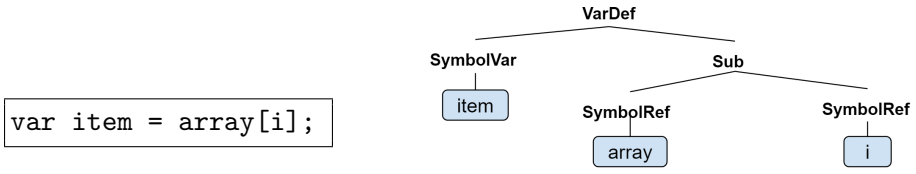
Next, we define AST pairwise paths. For convenience, in the rest of this section we assume that all definitions refer to a single AST $\langle N, T, X, s, \delta, val \rangle$.

An AST pairwise path is a path between two nodes in the AST, formally defined as follows:

Definition 2.3.2 (AST path)**.** An AST-path of length k is a sequence $n_1 d_1 ... n_k d_k n_{k+1}$, where for $i \in [1..k+1]$: $n_i \in (N \cup T)$ are terminals or nonterminals and for $i \in [1..k]$: $d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree). If $d_i = \uparrow$, then: $n_{i+1} = \pi(n_i)$; if $d_i = \downarrow$, then: $n_i = \pi(n_{i+1})$. We use $start(p)$ to denote n_1 and $end(p)$ to denote n_{k+1}.

We define a *path-context* as a tuple of an AST path and the values associated with its end nodes: (i.e. n_1 and n_{k+1}). In general, we consider path-contexts which span between arbitrary AST nodes, e.g., a terminal and its ancestor, but for simplicity, we base the following definitions on path-contexts which span between terminals:

Definition 2.3.3 (Path-context)**.** Given an AST Path p, its path-context is the triplet $\langle x_s, p, x_f \rangle$ where $x_s = val(start(p))$ and $x_f = val(end(p))$ are the values associated with the start and end nodes of p.



(a) (b)

Figure 2.3: A JavaScript statement and its partial AST.

That is, a path-context describes two nodes from the AST with the syntactic path between them.

Finally, we define an *Abstract path-context* as an abstraction of concrete path context:

Definition 2.3.4 (Abstract path-context)**.** Given a path-context $\langle x_s, p, x_f \rangle$ and an abstraction function $\alpha : P \rightarrow \hat{P}$, an abstract path-context is the triplet $\langle x_s, \alpha(p), x_f \rangle$, where P is the set of AST paths, $\hat{P}$ is the set of abstract AST paths, and α is a function that maps a path to an abstract representation of it.

The abstraction function α is any function that transforms a path to a different representation. A trivial abstraction function is α_{id}, which maps a path to itself: $\alpha_{id}(p) = p$.

Example 2.3.5. For example, consider the JavaScript line of code in Figure 2.3a and its partial AST in Figure 2.3b. We denote the path between the variable `item` to the variable `array` by p. Using α_{id}, the abstract path-context of p is:

$$\langle \texttt{item}, \alpha_{id}(p), \texttt{array} \rangle = \tag{2.1}$$

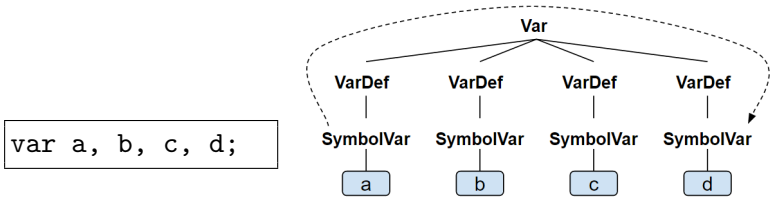
$$\langle \texttt{item}, (SymbolVar \uparrow VarDef \downarrow Sub \downarrow SymbolRef), \texttt{array} \rangle \tag{2.2}$$

Using a different abstraction function yields a different abstract path-context, for example $\alpha_{forget-arrows}$:

$$\langle \texttt{item}, \alpha_{forget-arrows}(p), \texttt{array} \rangle = \tag{2.3}$$

$$\langle \texttt{item}, (SymbolVar, VarDef, Sub, SymbolRef), \texttt{array} \rangle \tag{2.4}$$

Naïvely extracting all the paths in the AST and representing each of them uniquely can be computationally infeasible, and as a result of the bias-variance tradeoff (Hastie et al., 2001), can lead to worse prediction results. However, alternative abstraction functions can be used to control the number of distinct extracted paths. In Section 2.4.6 we describe alternative abstractions that abstract some of the information, and thus allow us to tune the trade-off between accuracy, training time, and model size.



(a) (b)

Figure 2.4: An example statement and its AST, with an example of a path between the *SymbolVar* terminals that represent `a` and `d`. The length of this path is 4, and its width is 3.

2.3.2 Limiting the Number of Paths

Another approach for controlling the number of distinct paths is to limit the *number of extracted paths*.

Path length and width We define hyper-parameters that limit the path length and width. We define the following hyper-parameters:

- *max_length*, defined as the maximal *length* of a path, i.e., the maximum value of k.
- *max_width*, defined as the maximal allowed difference between sibling nodes that participate in the same path, as shown in Figure 2.4.

When limiting these parameters to certain values, we do not extract longer or wider paths. We tune the optimal values of width and length by grid search of combinations on a validation set of programs and choose the combination that yields the highest accuracy, as described in Section 2.4. The tuning process of finding the optimal parameter values should be separate for each language and task.

Obviously, setting the values of these parameters to a value that is too low limits the expressiveness of the paths, does not capture enough context for each element, limits the ability to model the training and test data, and therefore produces poor accuracy. Why, then, does limiting the path length and width actually improve accuracy? There are several reasons:

- **Locality** The role of a program element is affected mostly by its surroundings. For example, consider the program in Figure 2.5. The width of a path that connects the variable `a` in the first line to the variable `b` in the last line is as large as the number of lines in the program. Usually, the names of `a` and `b` can be predicted by considering elements within a closer distance. Therefore, using paths between too distant elements can cause noise and pollute the relevant information.
- **Sparsity** Using paths that are too long can cause the representation space to be too sparse. A long path might appear too few times (or even only once) in the

```
assert.equal(a,1);
assert.equal(...);
...
assert.equal(b,1);
```

Figure 2.5: An example of a typical program where the maximal path length is relatively small, but the width can be large.

training set and cause the model to predict specific labels with high probability. This phenomenon is known as *overfitting*, where the learned AST paths are very specific to the training data and the model fails to generalize to new, unseen data.

- **Performance** There is a practical limit on the amount of data that a model can be trained on. Too much data can cause the training phase to become infeasibly long. There is a tradeoff between how many programs the model can be trained on, and how many paths are extracted from each program. Therefore, it makes sense to limit the number of extracted paths from each program by limiting the paths' length and width, in order to be able to train on a larger and more *varied* training set.

In fact, tuning path length and width is used to control the bias-variance tradeoff. Shorter paths increase the bias error, while longer paths increase the variance error. The relationship between these parameters and results is discussed and demonstrated in Section 2.4.5.

2.4 Evaluation

Since the goal of this work is to *provide a representation of program elements*, we compared the effect of different representations on the accuracy of the learning algorithms. To show that our approach can be applied to the representation of the input without modifying the learning algorithm, we used off-the-shelf learning algorithms but represented the input in each experiment using *a different representation* (when possible).

Our evaluation aims to answer the following questions:

- How useful are AST paths compared to existing representations? (Section 2.4.3)
- How useful are AST paths across different programming languages, tasks and learning algorithms? (Section 2.4.3)
- Do AST paths just memorize the input, or do they capture deeper semantic regularities? (Section 2.4.4)
- How long are the useful paths? How do the paths' length and width affect the results? (Section 2.4.5)

- How important is the concrete representation of paths? Which abstractions can be used to represent paths without reducing accuracy? (Section 2.4.6)

Leafwise and semi-paths Although the family of representations in this work includes n-wise paths and paths between any kind of AST nodes, for simplicity and feasible training time, we performed most of the experiments using leafwise-paths (paths between AST terminals) and semi-paths — paths between an AST terminal and one of its ancestor nodes in the AST. The idea is that leafwise-paths are more diverse and therefore more expressive than semi-paths, but semi-paths provide more *generalization*. Semi-paths allow us to generalize learning and capture common patterns in different programs, even if the full path does not recur.

An exception is the prediction of full types in Java, in which we predict types of expressions which are not necessarily terminals. In this case, we also used paths between terminals to the nonterminal in question.

2.4.1 Prototype Implementation

We implemented a cross-language tool called PIGEON. The tool consists of separate modules that parse and traverse the AST of a program in each different language, but the main algorithm is the same across all languages. Currently PIGEONcontains modules for Java, JavaScript, Python and C#, and it can be easily extended to any other language.

AST construction and path extraction For Java we used JavaParser; for JavaScript we used UglifyJS for parsing and traversing the AST, along with additional modifications from UnuglifyJS; for Python we used the Python internal parser and AST visitor; and for C# we used Roslyn.

Learning algorithms We experiment with two learning algorithms: Conditional Random Fields, based on the implementation of Nice2Predict (Raychev et al., 2015), and the word2vec based implementation of Levy and Goldberg (2014a).

To support our representation in the learning engine side and produce a qualitative evaluation, we introduced minor extensions to the Nice2Predict framework:

- *Support unary factors.* Previously, Nice2Predict supported only pairwise feature functions, and we implemented support for unary factors to express the relationship between different occurrences of the same identifier. Note that this is required because different AST nodes for the same identifier are merged into a single node in the CRF. Hence, a path between these nodes in the AST becomes a unary-factor in the CRF. This extension increases accuracy by about 1.5%.

- *Top-k candidates suggestion.* CRFs output a single prediction for each program element. We implemented an additional API that receives a parameter k and

Table 2.1: The amounts of data used for the experimental evaluation of each language.

Language	Total repos	Training Set File	Training Set Size (GB)	Test set File	Test set Size (MB)
Java	10,081	1,717,016	16	50,000	1001
JavaScript	6,863	159,038	3.4	38,103	130
Python	8,565	458,771	5.4	39,941	588
C#	1,000	262,774	4.7	50,000	1208

suggests the *top-k* candidate names for each program element (this extension was adopted into Nice2Predict). This allowed us to manually investigate the quality of results (Section 2.4.4). When all top-k predictions for a variable name captured similar notions, it increased our confidence that the model performs stable predictions.

2.4.2 Experimental Setting

Datasets For each language, we collected source code from public GitHub projects, and split it randomly to training, validation and test sets. Our data included the top ranked projects of each language and the projects that were forked the most. Table 2.1 shows the amount of data used for each language. Java required an order of magnitude more data than the other languages: we had to keep enlarging our Java dataset to achieve results that were close to the other languages.

Following recent work which found a large amount of code duplication in GitHub (Lopes et al., 2017), we devoted much effort into filtering duplicates from our dataset, and especially the JavaScript dataset. To filter duplicates, we used file names, directory names (such as "node_modules"), and md5 of files. In Java and Python, which do not commit dependencies, duplication is less severe (as also observed by Lopes et al. (2017)). Furthermore, in our setting, we took the top-ranked and most popular projects, in which we observed duplication to be less of a problem (Lopes et al. (2017) measured duplication across *all* the code in GitHub).

Evaluation metric For simplicity, in all the experiments we measured the percentage of exact match predictions, case-insensitive and ignoring differences in non-alphabetical characters. For example, this metric considers `totalCount` as an exact match to `total_count`. An exception is the comparison to Allamanis et al. (2016), who optimized their Java method name prediction model to maximize the F1 score over sub-tokens. In this case, we compared their model with ours on both exact match and F1 score. An unknown test label ("UNK") was always counted as an incorrect prediction, or as a possibly partial prediction when using the F1 score, and our model never suggests "UNK". For example, if the true test label is `get<UNK>`, our model could get partial precision and partial recall for predicting `getFoo`.

Table 2.2: Accuracy comparison for variable name prediction, method name prediction, and full type prediction using CRFs.

	Previous work		AST Paths (this work)
Task: Variable name prediction			
JavaScript	24.9% (no-paths)	60.0% (UnuglifyJS)	**67.3%**
Java	23.7% (rule-based)	50.1% (CRFs+4-grams)	**58.2%**
Python	35.2% (no-paths)		**56.7%** (top-7)
C#			**56.1%**
Task: Method name prediction			
JavaScript	44.1% (no-paths)		**53.1%**
Java	16.5%, F1: 33.9 (Allamanis et al. (2016))		**47.3%**, F1: **49.9**
Python	41.6% (no-paths)		**51.1%** (top-7)
Task: Full type prediction			
Java	24.1% (naïve baseline)		**69.1**%

Table 2.3: Accuracy comparison for the variable name prediction task that was evaluated using word2vec in JavaScript.

Model	Names Accuracy
linear token-stream + word2vec	20.6%
path-neighbors, no-paths + word2vec	23.2%
AST Paths (this work) + word2vec	**40.4%**

2.4.3 Quantitative Evaluation

We conducted several experiments to evaluate the usefulness of AST paths in different tasks and programming languages. We performed the following quantitative experiments:

- *Prediction of variable names across all four languages.* Variable names have sufficient training data in all languages to produce meaningful results. In this experiment we used both CRFs and word2vec. As baselines we used the work of Raychev et al. (2015), CRFs with token-based n-grams as factors, and a simple rule-based baseline. For JavaScript with word2vec, we used word2vec with linear token context as a baseline and show that path representations yield dramatic improvement.

- *Prediction of method names across JavaScript, Java and Python.* We compared our general approach for method name prediction with Allamanis et al. (2016), who used a convolutional neural network with attention.

- *Prediction of full types in Java.* For Java, we compared our results to a synthetic

(straw-man) baseline that predicts all types to be `java.lang.String`. This baseline shows that despite the prevalence of the `String` type, the task of type prediction is still very challenging.

In all of the following CRF experimental setups, "no-path" refers to a "bag-of-words" baseline, in which we used the same CRF learning algorithm, but used a single symbol to represent all relations. In this baseline, path information was hidden from the model during training and testing, and therefore it always assigned the same likelihood for each specific pair of identifiers, regardless of the syntactic relation between them. This baseline can be seen as a "bag of near identifiers" that uses the neighbors' names without their syntactic relation and therefore without considering the *way* program elements are related.

Predicting Variable Names

To predict variable names, we used both CRFs and word2vec.

Evaluation with CRFs We present our evaluation results with CRFs for names in the top part of Table 2.2. For JavaScript, where a tool that uses predefined features exists, we evaluated the other tool with the exact same datasets and settings, and the same AST terminals as CRF nodes, which makes the input representation (AST paths vs. their features) the only difference between the two experiments. Using our representations yields 7.6% higher accuracy than the previous work.

For Java, we compared the results with two baselines:

- *CRFs + n-grams* - this baseline uses the same CRF nodes as the path-based model, except that the relations between them are the *sequential* n-grams. We chose $n = 4$ as the value that maximizes accuracy on the validation set, such that the produced model consumes approximately the same amount of memory and disk as the path-based model.

- *Rule-based* - Since Java is a typed language which has a rich type system, and typical code tends to use a lot of classes and interfaces, we wonder whether the task of predicting variable names is easier in Java than in other languages and can be solved using traditional rule-based (non-learning) approaches. Our rule-based baseline predicts variable names based on the following pattern heuristics and statistics of the training corpus:

  - `for(int i = ...) {`
  - `this.<fieldName> = <fieldName>;`
  - `catch (... e) {`
  - `void set<fieldName>(... <fieldName>) {`
  - Otherwise: use the type: `HttpClient client`.

As shown, using CRFs with AST paths yields higher results than the baselines, in all the languages, showing that our representation yields higher results than manually defined features, n-grams, and rule-based approaches.

Evaluation with word2vec We present our evaluation results with a word2vec based implementation in Table 2.3. For comparison, we use two alternative approaches to represent the context for prediction:

- The *linear token-stream* approach uses the surrounding tokens to predict a variable name. Surrounding tokens (e.g., values, keywords, parentheses, dots and brackets) may implicitly hint at the syntactic relations, without AST paths. This is the type of context usually used in NLP, in the original implementation of word2vec, and in many works in programming languages.
- The *path-neighbors, no-paths* approach uses the same surrounding AST nodes for contexts as AST paths, except that the path itself is hidden, and only the identity of the surrounding AST nodes is used. The goal of using this baseline is to show that the advantage of AST paths over token-stream is not only in their wider *span*, but in the representation of the path itself.

Using word2vec with AST paths produces much better results compared to these baselines. This shows the advantage of using AST paths as context over token-stream based contexts, and the significance of using a representation of the paths for prediction.

Limitations of evaluation We noticed that our models often predict names that are very similar but not identical to the original name, such as `message` instead of `msg`, or synonyms such as `complete` instead of `done`; these are counted as incorrect predictions. Moreover, we noticed that our models sometimes predict *better* names than the original names. Therefore, the accuracy results are an underapproximation of the ability of AST paths to predict meaningful names.

Another limitation lies in the inability of CRFs and word2vec to predict out-of-vocabulary (OoV) names. As was previously observed (Allamanis et al., 2016, 2015a), there are two main types of OoV names in programs: names that did not appear in the training corpus but can be composed of known names (neologisms), and entirely new names. The total OoV rate among our various datasets and tasks varied between $5 - 15\%$, and specifically 7% for predicting variable names in JavaScript, and 13% for Java method names. Several techniques were suggested to deal with each type of OoV (Allamanis et al., 2016, 2015a), which we did not consider here and are out of scope of this work.

Discussion We note that the accuracy for Java is lower than for JavaScript. We have a few possible explanations: The JavaScript training set contains projects that are rather domain specific, mostly client and server code for web systems (for example, the

terms `request` and `response` are widely used across all projects). In contrast, the Java code is much more varied in terms of domains. Additionally, the Java naming scheme makes extensive use of compound names (e.g., `multithreadedHttpConnectionManager`), and this is amplified by the type-based name suggestions for variables provided by modern Java IDEs. In contrast, the JavaScript variable names are typically shorter and are not an amalgamation of multiple words (e.g., `value`, `name`, `elem`, `data` are frequent names).

The accuracy of C# is similar to Java, but using significantly less training data. We believe that C# naming is more structured because the commonly used C# IDE (VisualStudio), suggests variable names based on their types.

The accuracy for Python is lower than that of JavaScript. Manual examination of the training data shows that Python programs vary widely in code quality, making the training set more noisy than that of other languages. In addition, the variety of domains and IDEs for Python makes variable names less standard. Finally, Python is easy to write, even for non-programmers, and thus there is a wide variety of non-professional Python code. The low accuracy for Python is also consistent with Raychev et al. (2016a).

Comparison of CRFs and word2vec We observe that the accuracy of Pigeon + CRFs is higher than that of Pigeon+ word2vec, as can be seen in Table 2.2. One reason is that, unlike CRFs, word2vec was not designed exactly for this prediction task. Originally, word2vec was intended to produce meaningful word embeddings: given a set of query path-contexts, the vectors of all of them are assigned the same weight for predicting the unknown value.

Moreover, CRFs are relatively more interpretable. The weights assigned to each factor can be observed and explain a prediction posteriori. However, word2vec was faster to train and much more memory efficient. In our evaluation, the memory required for training was over 200GB for CRFs and only 10GB with word2vec. Further, the training time of CRFs was up to 100 hours, where word2vec required at most 5 hours.

The goal here is not to provide a fair comparison between CRFs and word2vec, as their prediction tasks are slightly different; our observations in this regard are merely anecdotal. The main goal is to compare *different representations for the same learning algorithm* and show that each of the learning algorithms separately can be improved by plugging in our simple representation.

Predicting Method Names

We present our evaluation results for predicting method names in Table 2.2. Accuracy was similar for all languages ($\sim 50\%$).

Good method names balance the need to describe the internal implementation of the method and its external usage (Høst and Østvold, 2009). For predicting method

```
def sh3(c):
    p = Popen(c, stdout=PIPE,
        stderr=PIPE, shell=True)
    o, e = p.communicate()
    r = p.returncode
    if r:
        raise CalledProcessError(r, c)
    else:
        return o.rstrip(), e.rstrip()
```

(a) Stripped names

```
def sh3(cmd):
    process = Popen(cmd, stdout=PIPE,
        stderr=PIPE, shell=True)
    out, err = process.communicate()
    retcode = process.returncode
    if retcode:
        raise CalledProcessError(retcode, cmd)
    else:
        return out.rstrip(), err.rstrip()
```

(b) AST paths + CRFs

Figure 2.6: Example of a Python program with stripped names and with predictions produced using our AST paths.

names, we use mostly the paths from within a method to its name, but when available in the same file, we also use paths from invocations of the method to the method name. Ideally, one would use paths from different files (and for library methods, even across projects), but this requires a non-local view, which we would like to avoid for efficiency reasons.

We use the internal paths from the leaf that represents the method name to other leaves within the method AST (which capture the method implementation) and the external paths from references of the method to their surrounding leaves (which represent the usage of the method). However, we observed that using only internal paths yields only 1% lower accuracy.

In Java, CRFs with AST paths are compared to the model of Allamanis et al. (2016), which we trained on the same training corpus. Since their model is optimized to maximize the F1 score over sub-tokens, Table 2.2 presents both exact accuracy and F1 score for method name prediction in Java. The table shows that CRFs with AST paths significantly improve over the previous work in both metrics.

Predicting Full Types

Our results for predicting full types in Java using CRFs are shown in the bottom part of Table 2.2. Our goal is to predict the full type even when it explicitly appears in the code (e.g., `com.mysql.jdbc.Connection`, rather than `org.apache.http.Connection`). Here we also use paths from leaves to nonterminals which represent expressions. The evaluated types were only those that could be solved by a global type inference engine. Therefore, accuracy is the percent of correct predictions out of the results that are given by type inference.

Although a type inference engine still produces more accurate results than our learning approach, our results using AST paths are surprisingly good, especially considering the relative simplicity of our representation. We also note that type inference is a global task, and our approach reconstructs types locally without considering the global scope of the project.

```
function f(a, b, c) {
  b.open('GET', a, false);
  b.send(c);
}
```

(a) Stripped Names

```
function f(url, request, callback) {
  request.open('GET', url, false);
  request.send(callback);
}
```

(b) AST Paths + CRFs

```
function f(source, req, n) {
  req.open("GET", source, false);
  req.send(n);
}
```

(c) `nice2predict.org`

Figure 2.7: Example of a JavaScript program with stripped names, with predictions produced using our AST paths and an online version of UnuglifyJS at nice2predict.org. This is the default example shown at nice2predict.org.

```
int count(List<Integer> x, int t) {
    int c = 0;
    for (int r: x) {
        if (r == t) {
            c++;
        }
    }
    return c;
}
```

(a) Stripped names

```
int count(List<Integer> values, int value) {
    int count = 0;
    for (int v: values) {
        if (v == value) {
            count++;
        }
    }
    return count;
}
```

(b) AST paths + CRFs

Figure 2.8: Examples of Java programs with stripped names and with predictions produced using our AST paths. We deliberately selected challenging examples in which the prediction cannot be aided by specific classes and interfaces.

CRFs with AST paths achieved 69.1% accuracy when predicting full type for Java. We contrast this result with a naïve baseline that uniformly predicts the type `java.lang.String` for all expressions. This naive baseline yields an accuracy of 24.1%, which shows the task is nontrivial, even when factoring out the most commonly used Java type.

2.4.4 Qualitative Evaluation

Our qualitative evaluation includes:

- An anecdotal study of name prediction in different languages. For JavaScript we also compared our predictions to those of Raychev et al. (2015) in interesting cases.
- An anecdotal study of top-k predictions for some examples, showing semantic similarities between predicted names as captured by the trained model.

Prediction Examples

Figure 2.6 shows an example of a Python program predicted using AST paths. It can be seen that all the names predicted using AST paths were renamed with meaningful names such as `process`, `cmd` and `retcode`.

Figure 2.7 shows the default JavaScript example from nice2predict.org, predicted using AST paths and an online version of UnuglifyJS at nice2predict.org. We note that their online model was not trained on the same dataset as our model. The model which was trained using UnuglifyJS and our dataset yielded worse results. It can be seen that our model produced more meaningful names such as `url` (instead of `source`) and `callback` (instead of `n`).

Figure 2.8 shows examples of Java programs. To demonstrate the expressiveness of AST paths, we deliberately selected challenging examples in which the prediction cannot be aided by the informative class and interface names that Java code usually contains (as in: `HttpClient client`). Instead, our model had to leverage the syntactic structure to predict the meaningful names: `done`, `values`, `value` and `count`.

2.4.5 Impact of Parameter Values

In Section 2.3 we introduced and discussed the importance of the *max_length* and *max_width* parameters. For each language we experimented with different combinations of values for *max_length* and *max_width* on its validation set. We chose the values that produced the highest accuracy while still being computationally feasible when evaluating the model with the test set.

Accuracy with different path length and width We experimented with tuning the path parameters and observed their effect on the accuracy. The best parameter values for each prediction are shown in Table 2.2.

For the task of name prediction, for all languages, the best path length is 6-7, and the best width is 3-4. The variations in path length stem from minor differences in the structure of the AST. For example, despite the similarity in source level between Java and C#, the C# AST is slightly more elaborate than the one we used for Java.

A drill-down of the accuracy given different parameter values for variable name prediction in JavaScript is shown in Figure 2.9. We observe that the *max_length* parameter has a significant positive effect, while the contribution of a larger *max_width* is positive but minor. This observation affirms our initial hypothesis that our long-distance paths are fundamental and crucial to the accuracy of the prediction. It also confirms our belief that an automatic representation of code (rather than manually defined) is essential, since the long-distance paths are very unlikely to have been designed manually.

For the task of method name prediction, since there are significantly fewer paths, we could afford to set a high parameter value without too much tuning and still keep

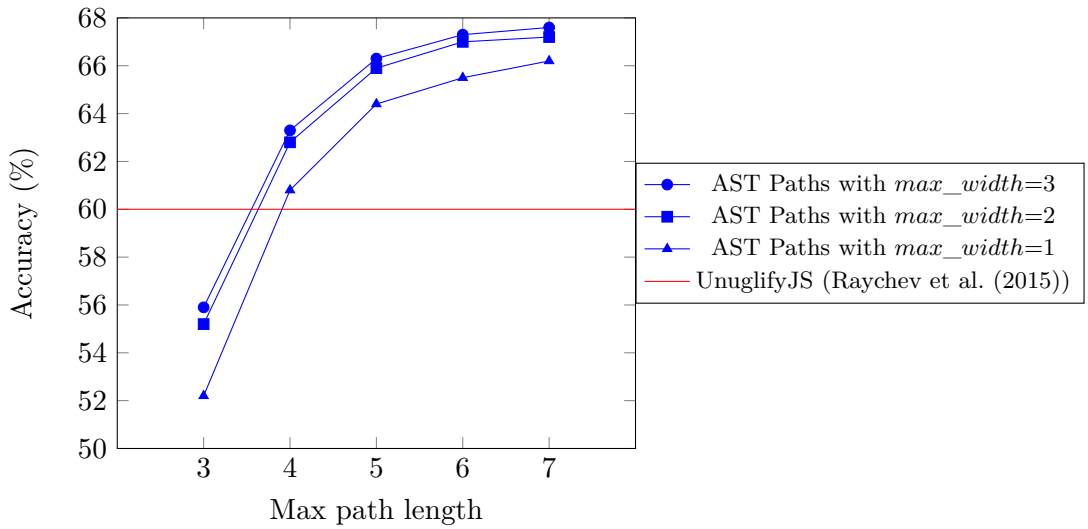


Figure 2.9: Accuracy results of AST paths with CRFs, for the task of variable naming in JavaScript, for different combination values of *max_length* and *max_width* (UnuglifyJS is presented here for comparison).

the training time and resources feasible. We therefore set the length in this case to 12 for JavaScript, 10 for Python, and just 6 for Java.

For the task of predicting full types in Java, we used length 4 and width 1, which yielded an accuracy of 69.1%. The intuition for the short path length is that in many cases the type of an expression can be inferred locally from other neighboring types, often from an explicit type declaration.

Higher values for *max_length* and *max_width* resulted in higher training times, but combined with the *downsampling* approach, it is possible to maintain a shorter training time while increasing the parameter values, and control the tradeoff between accuracy and training time.

2.4.6 Abstractions of AST Paths

In order to evaluate the full expressiveness of AST paths, the previously reported experiments were performed using no abstraction, i.e. α_{id}. However, it is also possible to use a higher level of abstraction. Instead of representing the whole path node-by-node with separating up and down arrows, it is possible to keep only parts of this representation. This abstraction results in less expressive paths and might represent two different paths as equal, but it enables decreasing the number of distinct paths, thus reducing the number of model parameters. Training will be faster as a result.

Different levels of path abstractions also allow us to evaluate the importance of different components of AST paths, or *which components of AST paths contribute to their usefulness the most*. We experimented with several levels of abstraction:

- "No-arrows" - using the full path encoding, except the up and down symbols $\{\uparrow, \downarrow\}$.

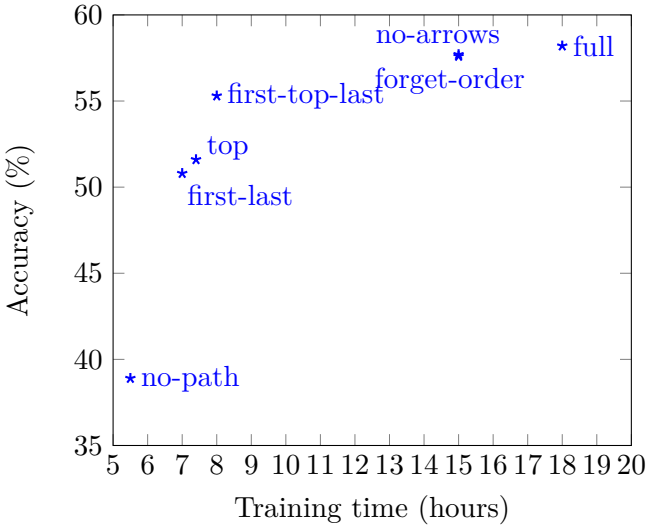


Figure 2.10: The accuracy of each abstraction method compared to the consumed training time, for the task of variable naming in Java

- "Forget-order" - using paths without arrows and without order between the nodes: instead of treating a path as a *sequence* of nodes, treat it as a *bag* of nodes.
- "First-top-last" - keeping only the first, top and last nodes of the path. The "top" node refers to the node that is hierarchically the highest, from which the direction of the path changes from upwards to downwards.
- "First-last" - keeping only the first and last nodes.
- "Top" - keeping only the top node.
- "No-paths" - using no paths at all, and treating all relations between program elements as the same. The name of an element is predicted by using the bag of surrounding identifiers, without considering the syntactic relation to each of them.

All of the following experiments were performed using CRFs for variable names prediction, on the Java corpus and on the same hardware. In every experiment, the training corpus and the rest of the settings were identical. The number of training iterations was fixed.

Figure 2.10 shows the accuracy of each abstraction compared to the consumed training time. As shown, as more information is kept, accuracy is increased, with the cost of a longer training time. An interesting "sweet-spot" is "first-top-last", which reduces training time by half compared to the full representation, with accuracy that is as 95% as good.

We also observe that the arrows and the order of the nodes in the path contribute about 1% accuracy.

Chapter 3

code2vec: Learning Distributed Representations of Code

3.1 Introduction

Distributed representations of words (such as "word2vec") (Mikolov et al., 2013a,b; Pennington et al., 2014), sentences, paragraphs, and documents (such as "doc2vec") (Le and Mikolov, 2014) have played a key role in unlocking the potential of neural networks for natural language processing (NLP) tasks (Bengio et al., 2003; Collobert and Weston, 2008; Socher et al., 2011; Turian et al., 2010; Glorot et al., 2011; Turney, 2006). Methods for learning distributed representations produce low-dimensional vector representations for objects, referred to as *embeddings*. In these vectors, the "meaning" of an element is distributed across multiple vector components, such that semantically similar objects are mapped to close vectors.

Goal: The goal of this work is to learn *code embeddings*, continuous vectors for representing snippets of code. By learning code embeddings, our long-term goal is to enable the application of neural techniques to a wide range of programming-language tasks. In this work, we use the motivating task of *semantic labeling of code snippets*.

Motivating task: semantic labeling of code snippets Consider the method in Figure 3.1. The method contains only low-level assignments to arrays, but a human reading the code may (correctly) label it as performing the *reverse* operation. Our goal is to predict such labels automatically. The right-hand side of Figure 3.1 shows the labels predicted automatically using our approach. The most likely prediction (77.34%) is *reverseArray*. Section 3.5 provides additional examples.

This problem is hard because it requires *learning a correspondence* between the *entire content of a method* and a semantic label. That is, it requires aggregating possibly hundreds of expressions and statements from the method body into a single, descriptive label.

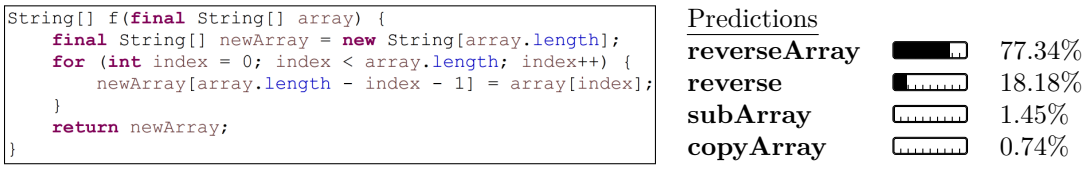


Figure 3.1: A code snippet and its predicted labels as computed by our model.

Our approach We present a novel framework for predicting program properties using neural networks. Our main contribution is a neural network that learns *code embeddings* — continuous distributed vector representations for code. The code embeddings allow us to model correspondence between code snippets and labels in a natural and effective manner.

Our neural network architecture uses a representation of code snippets that leverages the structured nature of source code and learns to aggregate multiple syntactic paths into a single vector. This ability is fundamental to the application of deep learning in programming languages, in the same way that word embeddings in natural language processing (NLP) are fundamental to the application of deep learning for NLP tasks.

The input to our model is a code snippet and a corresponding tag, label, caption, or name. This label expresses the semantic property that we wish the network to model, for example, a tag that should be assigned to the snippet, or the name of the method, class, or project that the snippet was taken from. Let $\mathcal{C}$ be the code snippet and $\mathcal{L}$ be the corresponding label or tag. Our underlying hypothesis is that *the distribution of labels can be inferred from syntactic paths in* $\mathcal{C}$. Our model therefore attempts to learn the label distribution, conditioned on the code: $P(\mathcal{L}|\mathcal{C})$.

We demonstrate the effectiveness of our approach for the task of predicting a method's name given its body. This problem is important as good method names make code easier to understand and maintain. A good name for a method provides a high-level summary of its purpose. Ideally, *"If you have a good method name, you don't need to look at the body."* (Fowler and Beck, 1999). Choosing good names can be especially critical for methods that are part of public APIs, as poor method names can doom a project to irrelevance (Allamanis et al., 2015a; Høst and Østvold, 2009).

Capturing semantic similarity between names During the process of learning code vectors, a parallel vocabulary of vectors of the labels is learned. When using our model to predict method names, the method-name vectors provide surprising semantic similarities and analogies. For example, $vector(\texttt{equals}) + vector(\texttt{toLowerCase})$ results in a vector that is closest to $vector(\texttt{equalsIgnoreCase})$.

Like the famous NLP example of: $vec(\text{“}king''\text{)} - vec(\text{“}man''\text{)} + vec(\text{“}woman''\text{)} \approx vec(\text{“}queen''\text{)}$ (Mikolov et al., 2013c), our model learns analogies that are relevant to source code, such as: *"`receive` is to `send` as `download` is to: `upload`"*. Table 3.1 shows additional examples, and Section 3.5.4 provides a detailed discussion.

Table 3.1: Semantic similarities between method names.

A	≈B
size	getSize, length, getCount, getLength
active	isActive, setActive, getIsActive, enabled
done	end, stop, terminate
toJson	serialize, toJsonString, getJson, asJson
run	execute, call, init, start
executeQuery	executeSql, runQuery, getResultSet
actionPerformed	itemStateChanged, mouseClicked, keyPressed
toString	getName, getDescription, getDisplayName
equal	eq, notEqual, greaterOrEqual, lessOrEqual
error	fatalError, warning, warn

3.1.1 Applications

Embedding a code snippet as a vector has a variety of machine-learning based applications, since machine-learning algorithms usually take vectors as their inputs. In this work, we examine the following direct applications:

1. *Automatic code review* - Suggesting better method names when the name given by the developer doesn't match the method's functionality. Better method names prevent naming bugs, improve the readability and maintenance of code, and facilitate the use of public APIs. This application was previously shown to be of significant importance (Fowler and Beck, 1999; Allamanis et al., 2015a; Høst and Østvold, 2009).

2. *Retrieval and API discovery* - Semantic similarities enable search in "the problem domain" instead of search "in the solution domain". For example, a developer might look for a `serialize` method, while the equivalent method of the class is named `toJson` as serialization is performed via `json`. An automatic tool that looks for the *vector* most similar to the requested name among the available methods will find `toJson` (Table 3.1). Such semantic similarities are difficult to find without our approach. Further, an automatic tool which uses our vectors can easily determine that a programmer is using the method `equals` right after `toLowerCase` and suggest using `equalsIgnoreCase` instead (Table 3.6).

The code vectors we produce can be used as input to any machine learning pipeline that performs tasks such as code retrieval, captioning, classification and tagging, or as a metric for measuring similarity between snippets of code for ranking and clone detection. The novelty of our approach is in its ability to produce vectors that capture properties of snippets of code, such that similar snippets (according to any desired criteria) are assigned similar vectors. This ability unlocks a variety of applications for working with machine-learning algorithms on code.

We deliberately picked the difficult task of method name prediction, for which prior results were poor (Allamanis et al., 2016; Alon et al., 2018; Allamanis et al., 2015a), as an evaluation benchmark. Succeeding in this challenging task implies good performance in other tasks such as predicting whether or not a program performs I/O, predicting the required dependencies of a program, and predicting whether a program is a suspected malware. We show that even for this challenging benchmark, our technique dramatically improves the results of previous works.

3.1.2 Challenges: Representation and Attention

Assigning a semantic label to a code snippet (such as a name to a method) is an example for a class of problems that require a compact semantic descriptor of a snippet. The question is how to represent code snippets in a way that captures some semantic information, is reusable across programs, and can be used to predict properties such as a label for the snippet. This leads to two challenges:

- Representing a snippet in a way that enables learning across programs.
- Learning which parts in the representation are relevant to prediction of the desired property, and learning the order of importance of the part.

Representation NLP methods typically treat text as a linear sequence of tokens. Indeed, many existing approaches also represent source code as a token stream (Allamanis et al., 2014; Allamanis and Sutton, 2013; Allamanis et al., 2016; Movshovitz-Attias and Cohen, 2013; White et al., 2015; Hindle et al., 2012). However, as observed previously (Bielik et al., 2016; Raychev et al., 2015; Alon et al., 2018), programming languages can greatly benefit from representations that leverage the structured nature of their syntax.

We note that there is a tradeoff between the degree of program analysis required to extract the representation and the learning effort that follows. Performing no program analysis but learning instead from the program's surface text often incurs a significant learning effort. This learning effort thus requires prohibitive amounts of data because the learning model has to re-learn the syntax and semantics of the programming language from the data. On the other end of the spectrum, performing a deep program analysis to extract the representation may make the learned model language-specific (and even task-specific).

Following previous works (Alon et al., 2018; Raychev et al., 2015), we use paths in the program's abstract syntax tree (AST) as our representation. By representing a code snippet using its syntactic paths, we can capture regularities that reflect common code patterns. We find that this representation significantly lowers the learning effort (compared to learning over program text), and is still scalable and general such that it can be applied to a wide range of problems and large amounts of code.

We represent a given code snippet as a bag (multiset) of its extracted paths. The challenges are then *how to aggregate a bag of contexts and which paths to focus on for making a prediction.*

Attention The problem can be stated informally as the need to learn a correspondence between a bag of path-contexts and a label. Representing each bag of path-contexts *monolithically* will result in sparsity – even similar methods will not have the *exact* same bag of path-contexts. We therefore need a *compositional* mechanism that can aggregate a bag of path-contexts such that bags that yield the same label are mapped to close vectors. Such a compositional mechanism would be able to generalize and represent new unseen bags by utilizing the individual path-contexts and their components (paths, values, etc.) that were observed during training to be parts of other bags.

To address this challenge we use a novel neural attention network architecture. Attention models have gained much popularity recently, mainly for neural machine translation (NMT) (Bahdanau et al., 2014; Luong et al., 2015; Vaswani et al., 2017), reading comprehension (Levy et al., 2017; Seo et al., 2016), speech recognition (Chorowski et al., 2015; Bahdanau et al., 2016) and computer vision (Xu et al., 2015; Mnih et al., 2014; Ba et al., 2014).

Our neural attention mechanism learns how much focus ("attention") should be given to each element in a bag of path-contexts. It allows us to precisely aggregate the information captured in each individual path-context into a single vector that captures information about the entire code snippet. As we show in Section 3.5.4, our model is relatively interpretable: the weights allocated by our attention mechanism can be visualized to understand the relative importance of each path-context in a prediction. The attention mechanism is *learned simultaneously with the embeddings, optimizing both the atomic representations of paths and the ability to compose multiple contexts into a single code vector.*

Soft and hard attention The terms "soft" and "hard" attention were proposed for the task of image caption generation by Xu et al. (2015). Applied in our setting, *soft attention* means that weights are distributed "softly" over all path-contexts in a code snippet, while *hard attention* refers to selection of a single path-context to focus on at a time. The use of *soft attention* over syntactic paths is the main understanding that leads to the improved results. We compare our model to an equivalent model that uses hard attention in Section 3.5.2, and show that *soft attention* is more efficient for modeling code.

3.1.3 Existing Techniques

The problem of predicting program properties by learning from big code has seen great interest and progress in recent years (Raychev et al., 2016a; Allamanis et al., 2014;

Bielik et al., 2016; Allamanis and Sutton, 2013; Hindle et al., 2012). The ability to predict semantic properties of a program without running it, and with little or no semantic analysis at all, is crucial to a wide range of applications: predicting names for program entities (Alon et al., 2018; Raychev et al., 2015; Allamanis et al., 2015a), code completion (Raychev et al., 2014; Mishne et al., 2012), code summarization (Allamanis et al., 2016), code generation (Murali et al., 2018; Maddison and Tarlow, 2014; Amodio et al., 2017), and more (see (Allamanis et al., 2017; Vechev and Yahav, 2016) for a survey).

3.1.4 Contributions

The main contributions of this work are:

- A path-based attention model for learning vectors for arbitrary-sized snippet of code. This model allows us to embed a program, which is a discrete object, into a continuous space, such that it can be fed into a deep learning pipeline for various tasks.
- As a benchmark for our approach, we perform a quantitative evaluation for predicting cross-project method names, trained on more than 12M methods of real-world data and compared with previous works. Experiments show that our approach achieves significantly better results than previous works, which used Long Short-Term Memory networks (LSTMs), CNNs and CRFs.
- A qualitative evaluation that interprets the attention that the model has learned to give to the different path-contexts when making predictions.
- A collection of method name embeddings, which often assign semantically similar names to similar vectors, and even make it possible to compute analogies using simple vector arithmetic.
- An analysis that shows the significant advantages in terms of generalization ability and space complexity of our model, compared to previous non-neural works such as Alon et al. (2018) and Raychev et al. (2015).

3.2 Overview

In this section we demonstrate how our model assigns different vectors to similar snippets of code, *in a way that captures the subtle differences between them.* The vectors are useful for making a prediction about each snippet, even though none of these snippets has been observed in its entirely in the training data.

The main idea of our approach is to extract syntactic paths from within a code snippet, represent them as a bag of distributed vector representations, and use an attention mechanism to compute a learned weighted average of the path vectors in

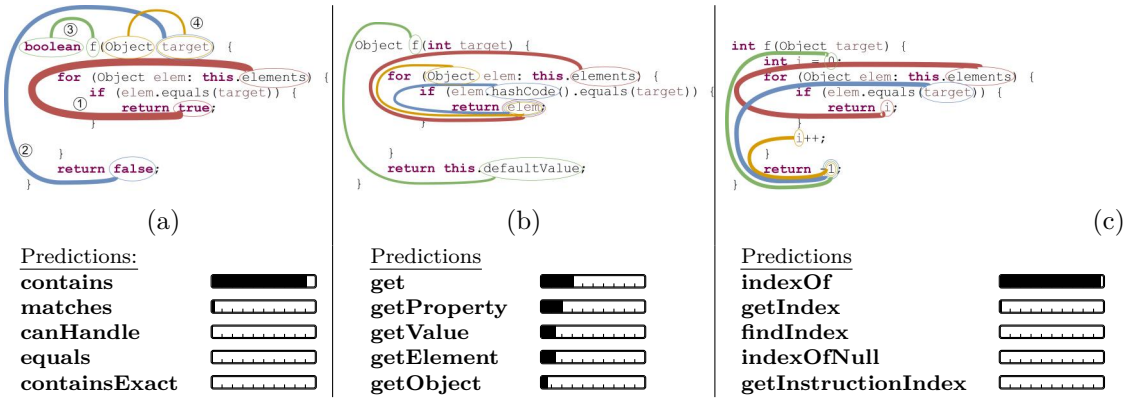


Figure 3.2: Examples of three methods that can be easily distinguished by our model despite having similar syntactic structure: our model successfully captures the subtle differences between them and predicts meaningful names. Each method portrays the top-4 paths that were given the most attention by the model. The widths of the colored paths are proportional to the attention that each path was given.

order to produce a single *code vector*. Finally, this code vector can be used for various tasks, such as to predict a likely name for the whole snippet.

3.2.1 Motivating Example

Since method names are usually descriptive and accurate labels for code snippets, we demonstrate our approach for the task of learning code vectors for method bodies and predicting the method name given the body. In general, the same approach can be applied to any snippet of code that has a corresponding label.

Consider the three Java methods in Figure 3.2. These methods share a similar syntactic structure. They all: have a single parameter named `target`; iterate over a field named `elements`; and have an `if` condition inside the loop body.

The main differences are that the method of Figure 3.2a returns `true` when `elements` *contains* `target` and `false` otherwise; the method of Figure 3.2b returns the element from `elements` for which `target` *equals its* `hashCode`; and the method of Figure 3.2c returns the *index of* `target` in `elements`. Despite their shared characteristics, our model captures the subtle differences and predicts the respective descriptive method names: `contains`, `get`, and `indexOf`.

Path extraction First, each query method in the training corpus is parsed to construct an AST. Then, the AST is traversed and syntactic paths between AST leaves are extracted. Each path is represented as a sequence of AST nodes, linked by up and down arrows, which symbolize the up or down link between adjacent nodes in the tree. The path composition is kept with the values of the AST leaves it is connecting, as a tuple we refer to as a *path-context*. These terms are defined formally in Section 3.3. Figure 3.3 portrays the top-four path-contexts that were given the most attention by

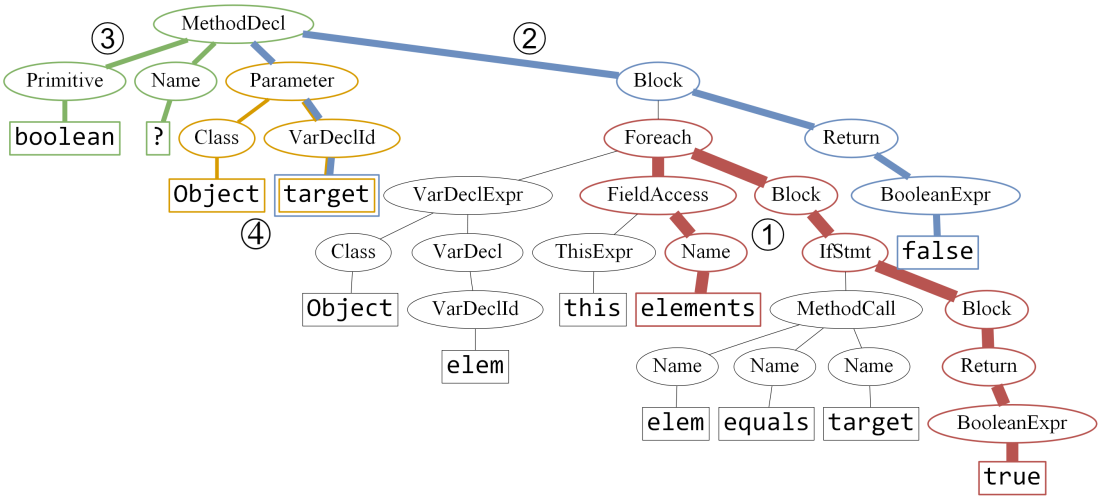


Figure 3.3: The top-4 attended paths of Figure 3.2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (**red** ①: **0.23**, **blue** ②: **0.14**, **green** ③: **0.09**, **orange** ④: **0.07**).

the model, on the AST of the method from Figure 3.2a, such that the width of each path is proportional to the attention it was given by the model during this prediction.

Distributed representation of contexts Each of the path and leaf-values of a path-context is mapped to its corresponding real-valued vector representation, or its *embedding*. Then, the three vectors of each context are concatenated to a single vector that represents that path-context. During training, the values of the embeddings are learned jointly with the attention parameter and the rest of the network parameters.

Path-attention network The Path-Attention network aggregates multiple path-context embeddings into a single vector that represents the entire method body. Attention is the mechanism that learns to score each path-context, such that higher attention is reflected in a higher score. These multiple embeddings are aggregated using the attention scores into a single *code vector*. The network then predicts the probability for each target method name given the code vector. The network architecture is described in Section 3.4.

Path-attention interpretation While it is usually difficult or impossible to interpret specific values of vector components in neural networks, it is possible and interesting to observe the attention scores that each path-context was given by the network. Each code snippet in Figure 3.2 and Figure 3.3 highlights the top-four path-contexts that were given the most weight (attention) by the model in each example. The widths of the paths are proportional to the attention score that each of these path-contexts was given. The model has learned how much weight to give every possible path on its

own, as part of training on millions of examples. For example, it can be seen in Figure 3.3 that the **red** ① path-context, which spans from the field `elements` to the return value `true`, was given the highest attention. For comparison, the **blue** ② path-context, which spans from the parameter `target` to the return value `false`, was given a lower attention.

Consider the **red** ① path-context of Figure 3.2a and Figure 3.3. As we explain in Section 3.3, this path is represented as:

(`elements`, Name↑FieldAccess↑Foreach↓Block↓IfStmt↓Block↓Return↓BooleanExpr, `true`)

Inspecting this path node-by-node reveals that this single path captures the main functionality of the method: the method iterates over a field called `elements`, and for each of its values it checks an `if` condition; if the condition is true, the method returns `true`. Since we use soft attention, the final prediction takes into account other paths as well, such as paths that describe the `if` condition itself, but it can be understood why the model gave this path the highest attention.

Figure 3.2 also shows the top-5 suggestions from the model for each method. As can be seen in all three examples, most of the top suggestions are very similar to each other and all of them are descriptive of the method. Observing the top-5 suggestions in Figure 3.2a shows that two of them (`contains` and `containsExact`) are very accurate, but it can also be imagined how a method called `matches` would share similar characteristics: a method called `matches` is also likely to have an `if` condition inside a `for` loop, and to return `true` if the condition is true.

Another interesting observation is that the **orange** ④ path-context of Figure 3.2a, which spans from `Object` to `target`, was given a lower attention than other path-contexts in the same method but *higher attention than the same path-context in Figure 3.2c.* This demonstrates how attention is not constant but is given with respect to the other path-contexts in the code.

Comparison with n-grams The method in Figure 3.2a shows the four path-contexts that were given the most attention during the prediction of the method name `contains`. Out of them, the **orange** ④ path-context, spans between two consecutive tokens: `Object` and `target`. This might create the (false) impression that representing this method as a bag-of-bigrams could be as expressive as syntactic paths. However, as can be seen in Figure 3.3, the **orange** ④ path goes through an AST node of type *Parameter*, which uniquely distinguishes it from, for example, a local variable declaration of the same name and type. In contrast, a bigram model will represent the expression `Object target` equally whether `target` is a method parameter or a local variable. This shows that a model using a syntactic representation of a code snippet can distinguish between two snippets of code that other representations cannot. By aggregating all the contexts using attention, the model can use subtle differences between

snippets to produce a more accurate prediction.

Key aspects The illustrated examples highlight several key aspects of our approach:

- A code snippet can be efficiently represented as a bag of path-contexts.
- Using a single context is not enough to make an accurate prediction. An attention-based neural network can identify the importance of multiple path-contexts and aggregate them accordingly to make a prediction.
- Subtle differences between code snippets are easily distinguished by our model, even if the code snippets have a similar syntactic structure and share many common tokens and n-grams.
- Large corpus, cross-project prediction of method names is possible using this model.
- Although our model is based on a neural network, the model is human-interpretable and provides interesting observations.

3.3 Background - Representing Code using AST Paths

In this section, we briefly describe the representation of a code snippet as a set of syntactic paths in its abstract syntax tree (AST). This representation is based on the general-purpose approach for representing program elements by Alon et al. (2018). The main difference in this definition is that we define this representation to handle *whole snippets of code*, rather than a single program element (such as a single variable), and use it as input to our path-attention neural network.

We start by defining an AST, a path and a path-context.

Definition 3.3.1 (Abstract Syntax Tree)**.** An Abstract Syntax Tree (AST) for a code snippet $\mathcal{C}$ is a tuple $\langle N, T, X, s, \delta, \phi \rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \to (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children, and $\phi : T \to X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.

Next, we define AST paths. For convenience, in the rest of this section we assume that all definitions refer to a single AST $\langle N, T, X, s, \delta, \phi \rangle$.

An AST path is a path between nodes in the AST, starting from one terminal, ending in another terminal, and passing through an intermediate nonterminal in the path which is a common ancestor of both terminals. More formally:

Definition 3.3.2 (AST path)**.** An AST-path of length k is a sequence of the form: $n_1 d_1 ... n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals, for $i \in [2..k]$: $n_i \in N$ are nonterminals and for $i \in [1..k]$: $d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree). If $d_i = \uparrow$, then: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. For an AST-path p, we use $start(p)$ to denote n_1 — the starting terminal of p, and $end(p)$ to denote n_{k+1} — its final terminal.

Using this definition we define a *path-context* as a tuple of an AST path and the values associated with its terminals:

Definition 3.3.3 (Path-context)**.** Given an AST Path p, its path-context is a triplet $\langle x_s, p, x_t \rangle$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$ are the values associated with the start and end terminals of p.

That is, a path-context describes two actual tokens with the syntactic path between them.

Example 3.3.4. A possible path-context that represents the statement: "`x = 7;`" would be:

$$\langle \texttt{x}, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), \texttt{7} \rangle$$

To limit the size of the training data and reduce sparsity, it is possible to limit different parameters of the paths. Following earlier works, we limit the paths by maximum *length* — the maximal value of k, and limit the maximum *width* — the maximal difference in child index between two child nodes of the same intermediate node. These values are determined empirically as hyperparameters of our model.

3.4 Model

In this section we describe our model in detail. Section 3.4.1 describes the way the input source code is represented, Section 3.4.2 describes the architecture of the neural network, Section 3.4.3 describes the training process, and Section 3.4.4 describes the way the trained model is used for prediction. Finally, Section 3.4.5 discusses some of the model design choices and compares the architecture to prior art.

High-level view At a high-level, the key point is that a code snippet is composed of a bag of contexts, and each context is represented by a vector whose values are learned. The values of this vector capture two distinct notions: the semantic meaning of this context, and the amount of attention this context should get.

The problem is as follows: given an arbitrarily large number of context vectors, we need to aggregate them into a single vector. Two trivial approaches would be to learn the most important one of them, or to use them all by vector-averaging them. These

alternatives will be discussed in Section 3.5.2, and the results of implementing them are shown in Table 3.4 ("hard attention" and "no-attention") to yield poor results.

Our main insight in this work is that *all* context vectors should be used but the model should be allowed to learn how much focus to give each vector. This is done by learning how to average context vectors in a weighted manner. The weighted average is obtained by weighting each vector by a factor of its dot product with another global attention vector. The vector of each context and the global attention vector are trained and learned *simultaneously* using the standard neural approach of backpropagation. Once trained, the neural network is simply a pure mathematical function, which uses algebraic operators to output a code vector given a set of contexts.

3.4.1 Code as a Bag of Path-Contexts

Our path-attention model receives as input a code snippet in some programming language and a parser for that language.

Representing a snippet of code We denote by Rep the representation function (also known as a feature function) which transforms a code snippet into a mathematical object that can be used in a learning model. Given a code snippet $\mathcal{C}$ and its AST $\langle N, T, X, s, \delta, \phi \rangle$, we denote by $TPairs$ the set of all pairs of AST terminal nodes (excluding pairs that contain a node and itself):

$$TPairs\left(C\right) = \{(term_i, term_j) \,|term_i, term_j \in termNodes\left(C\right) \wedge i \neq j\}$$

where $termNodes$ is a mapping between a code snippet and the set of terminal nodes in its AST. We represent $\mathcal{C}$ as the set of path-contexts that can be derived from it:

$$Rep\left(\mathcal{C}\right) = \left\{ (x_s, p, x_t) \;\middle|\; \begin{array}{l} \exists (term_s, term_t) \in TPairs\left(\mathcal{C}\right) : \\ x_s = \phi\left(term_s\right) \wedge x_t = \phi\left(term_t\right) \\ \wedge\, start(p) = term_s \wedge end(p) = term_t \end{array} \right\}$$

That is, $\mathcal{C}$ is represented as the set of triplets $\langle x_s, p, x_t \rangle$ such that x_s and x_t are values of AST terminals, and p is the AST path that connects them. For example, the representation of the code snippet from Figure 3.2a contains, among others, the four AST paths of Figure 3.3.

3.4.2 Path-Attention Model

Overall, the model learns the following components: embeddings for paths and names (matrices *path_vocab* and *value_vocab*), a fully connected layer (matrix W), attention vector ($\boldsymbol{a}$), and embeddings for the tags (*tags_vocab*). We describe our model from left-to-right (Figure 3.4). We define two embedding vocabularies: *value_vocab* and *path_vocab*, which are matrices in which every row corresponds to an embedding

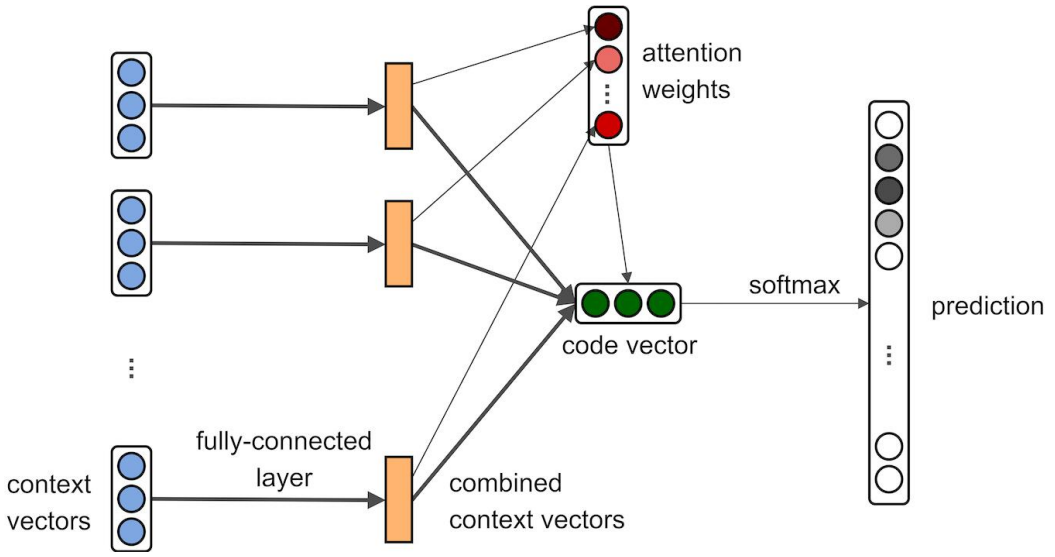


Figure 3.4: The architecture of our path-attention network. A *fully connected layer* learns to combine embeddings of each path-context with itself; attention weights are learned using the combined context vectors and used to compute a *code vector*. The code vector is used to predict the label.

associated with a certain object:

$$value\_vocab \in \mathbb{R}^{|X| \times d}$$
$$path\_vocab \in \mathbb{R}^{|P| \times d}$$

where as before, X is the set of values of AST terminals that were observed during training, and P is the set of AST paths. An embedding is looked up by simply picking the appropriate row of the matrix. For example, if we consider Figure 3.2a again, *value_vocab* contains rows for each token value such as `boolean`, `target` and `Object`. *path_vocab* contains rows which are mapped to each of the AST paths of Figure 3.3 (without the token values), such as the **red** ① path: $Name \uparrow FieldAccess \uparrow Foreach \downarrow Block \downarrow IfStmt \downarrow Block \downarrow Return \downarrow BooleanExpr$. The values of these matrices are initialized randomly and are learned simultaneously with the network during training.

The width of the matrix W is the embedding size $d \in \mathbb{N}$ – the dimensionality hyper-parameter. d is determined empirically, limited by the training time, model complexity, and the GPU memory, and it typically ranges between 100-500. For convenience, we refer to the embeddings of both the paths and the values as vectors of the same size d, but in general they can be of different sizes.

A bag of path-contexts $\mathcal{B} = \{b_1, ..., b_n\}$ that were extracted from a given code snippet is fed into the network. Let $b_i = \langle x_s, p_j, x_t \rangle$ be one of these path-contexts, such that $\{x_s, x_t\} \in X$ are values of terminals and $p_j \in P$ is their connecting path. Each component of a path-context is looked up and mapped to its corresponding embedding. The three embeddings of each path-context are concatenated to a single *context vector*:

$\boldsymbol{c_i} \in \mathbb{R}^{3d}$ that represents that path-context:

$$\boldsymbol{c_i} = embedding\left(\langle x_s, p_j, x_t \rangle\right) = \left[\ value\_vocab_s\ ;\ path\_vocab_j\ ;\ value\_vocab_t\ \right] \in \mathbb{R}^{3d} \tag{3.1}$$

For example, for the **red** ① path-context from Figure 3.3, its context vector would be the concatenation of the vectors of `elements`, the **red** ① path, and `true`.

Fully connected layer Since every context vector $\boldsymbol{c_i}$ is formed by a concatenation of three independent vectors, a fully connected layer learns to *combine* its components. This is done separately for each context vector, using the same learned combination function. This allows the model to give a different attention to every *combination* of paths and values. This combination allows the model the expressivity of giving a certain path more attention when observed with certain values and less attention when the exact same path is observed with other values.

Here, $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ is the output of the fully connected layer, which we refer to as a *combined context vector*, computed for a path-context b_i. The computation of this layer can be described simply as:

$$\tilde{\boldsymbol{c}}_{\boldsymbol{i}} = tanh\left(W \cdot \boldsymbol{c_i}\right)$$

where $W \in \mathbb{R}^{d \times 3d}$ is a learned weights matrix and *tanh* is the hyperbolic tangent function. The height of the weights matrix W determines the size of $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$, and for convenience is the same size (d) as before. In general, the height of W can be different; this will affect the size of the final *code* vector. *tanh* is the hyperbolic tangent element-wise function, a commonly used monotonic nonlinear activation function which outputs values in the range $(-1, 1)$, which increases the expressiveness of the model. That is, the fully connected layer "compresses" a context vector of size $3d$ into a combined context vector of size d by multiplying it with a weights matrix, and then it applies the *tanh* function to each element of the vector separately.

Aggregating multiple contexts into a single vector representation with attention The attention mechanism computes a weighted average over the combined context vectors, and its main job is to compute a scalar weight for each of them. An attention vector $\boldsymbol{a} \in \mathbb{R}^d$ is initialized randomly and learned simultaneously with the network. Given the combined context vectors: $\{\tilde{\boldsymbol{c}}_{\boldsymbol{1}}, ..., \tilde{\boldsymbol{c}}_{\boldsymbol{n}}\}$, the attention weight α_i of each $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ is computed as the normalized inner product between the combined context vector and the global attention vector $\boldsymbol{a}$:

$$\text{attention weight } \alpha_i = \frac{\exp(\tilde{\boldsymbol{c}}_{\boldsymbol{i}}^T \cdot \boldsymbol{a})}{\sum_{j=1}^{n} \exp(\tilde{\boldsymbol{c}}_{\boldsymbol{j}}^T \cdot \boldsymbol{a})}$$

The exponents in the equations are used only to make the attention weights positive, and they are divided by their sum to have a sum of 1, as a standard softmax function.

The aggregated code vector $\boldsymbol{v} \in \mathbb{R}^d$, which represents the whole code snippet, is a linear combination of the combined context vectors $\{\tilde{\boldsymbol{c}}_1, ..., \tilde{\boldsymbol{c}}_n\}$ factored by their attention weights:

$$\text{code vector } \boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot \tilde{\boldsymbol{c}}_i \tag{3.2}$$

That is, the attention weights are non-negative and their sum is 1, and they are used as the factors of the combined context vectors $\tilde{\boldsymbol{c}}_i$. Thus, attention can be viewed as a weighted average, where the weights are learned and calculated with respect to the other members in the bag of path-contexts.

Prediction Prediction of the tag is performed using the code vector. We define a tag vocabulary which is learned as part of training:

$$tags\_vocab \in \mathbb{R}^{|Y| \times d}$$

where Y is the set of tag values found in the training corpus. As before, the embedding of tag_i is row i of $tags\_vocab$. For example, looking at Figure 3.2a again, we see that $tags\_vocab$ contains rows for each of `contains`, `matches` and `canHandle`. The predicted distribution of the model $q(y)$ is computed as the (softmax-normalized) dot product between the code vector $\boldsymbol{v}$ and each of the tag embeddings:

$$for\, y_i \in Y : \; q(y_i) = \frac{\exp(\boldsymbol{v}^T \cdot tags\_vocab_i)}{\sum_{y_j \in Y} \exp(\boldsymbol{v}^T \cdot tags\_vocab_j)}$$

That is, the probability that a specific tag y_i should be assigned to the given code snippet $\mathcal{C}$ is the normalized dot product between the vector of y_i and the code vector $\boldsymbol{v}$.

3.4.3 Training

To train the network we use cross-entropy loss (Rubinstein, 1999, 2001) between the predicted distribution q and the "true" distribution p. Since p is a distribution that assigns a value of 1 to the actual tag in the training example and 0 otherwise, the cross-entropy loss for a single example is equivalent to the negative log-likelihood of the true label, and can be expressed as:

$$\mathcal{H}(p||q) = -\sum_{y \in Y} p(y) \, log \, q(y) = -log \, q(y_{true})$$

where y_{true} is the actual tag that was seen in the example. That is, the loss is the negative logarithm of $q(y_{true})$, the probability that the model assigns to y_{true}. As $q(y_{true})$ tends to 1, the loss approaches zero. The further $q(y_{true})$ goes below 1, the greater the loss becomes. Thus, minimizing this loss is equivalent to maximizing the log-likelihood that the model assigns to the true labels y_{true}.

The network is trained using any gradient descent based algorithm and the standard approach of back-propagating the training error through each of the learned parameters (i.e., deriving the loss with respect to each of the learned parameters and updating the learned parameter's value by a small "step" towards the direction that minimizes the loss).

3.4.4 Using the Trained Network

A trained network can be used to perform a downstream task, using the code vector $\boldsymbol{v}$ itself, and predict tags for new, unseen code.

Using the code vector An unseen code can be fed into the trained network exactly as in the training step, up to the computation of the code vector (Eq. equation 3.2). This code embedding can now be used in another deep learning pipeline for various tasks such as finding similar programs, code search, refactoring suggestion, and code summarization.

Predicting tags and names The network can also be used to predict tags and names for unseen code. In this case we also compute the code vector $\boldsymbol{v}$ using the weights and parameters that were learned during training, and prediction is done by finding the closest target tag:

$$prediction\,(\mathcal{C}) = argmax_{\mathcal{L}} P(\mathcal{L}|\mathcal{C}) = argmax_{\mathcal{L}}\left\{q_{v_{\mathcal{C}}}(y_{\mathcal{L}})\right\}$$

where $q_{v_{\mathcal{C}}}$ is the predicted distribution of the model, given the code vector $\boldsymbol{v_{\mathcal{C}}}$.

Scenario-dependant variants For simplicity, we describe a network that predicts a single label, but the same architecture can be adapted for slightly different scenarios. For example, in a multi-tagging scenario (Tsoumakas and Katakis, 2006), each code snippet contains multiple true tags as in StackOverflow questions. Another example is predicting a sequence of target words such as in method documentation. In the latter case, the attention vector should be used to re-compute the attention weights after each predicted token, given the previous prediction, as is commonly done in neural machine translation (Bahdanau et al., 2014; Luong et al., 2015).

3.4.5 Design Decisions

Bag of contexts We represent a snippet of code as an unordered bag of path-contexts. This choice reflects our hypothesis that the *existence* of path-contexts in a method body is more significant than their internal location or order.

An alternative representation is to sort path-contexts according to a predefined order (e.g., order of their occurrence). However, unlike natural language, there is no predetermined location in a method where the main attention should be focused. An

important path-context can appear anywhere in a method body (and span throughout the method body).

Working with syntactic-only context The main contribution of this work is its ability to aggregate multiple contexts into a fixed-length vector in a weighted manner and use the vector to make a prediction. In general, our proposed model is not bound to any specific representation of the input program; it can be applied in a similar way to a "bag of contexts" in which the contexts are designed for a specific task, or it can be applied to contexts that were produced using semantic analysis. Specifically, we chose to use a syntactic representation that is similar to that of Alon et al. (2018) because it was shown to be useful as a representation for modeling programming languages in machine learning models. It was also shown to be more expressive than n-grams and manually designed features.

An alternative approach is to include semantic relations as context. Such an approach was taken by Allamanis et al. (2018), who presented a Gated Graph Neural Network in which program elements are graph nodes and semantic relations such as `ComputedFrom` and `LastWrite` are edges in the graph. In their work, these semantic relations were chosen and implemented for specific programming language and tasks. In our work, we wish to explore *how far a syntactic-only approach can go*. Using semantic knowledge has many advantages and might reveal information that is not clearly expressed in a syntactic-only observation. However, using semantic knowledge comes at a cost: (i) an expert is required to choose and design the semantic analyses; (ii) generalizing to new languages is much more difficult, as the semantic analyses need to be implemented differently for every language; and (iii) the designed analyses might not easily generalize to other tasks. In contrast, in our syntactic approach (i) neither expert knowledge of the language nor manual feature designing is required; (ii) generalizing to other languages is accomplished by simply replacing the parser and extracting paths from the new language's AST using the same traversal algorithm; and (iii) the same syntactic paths generalize surprisingly well to other tasks (as was shown by Alon et al. (2018)).

Large corpus, simple model As Mikolov et al. (2013a) found for word representations, we found that a simpler model with a large amount of data is more efficient than a complex model and a small corpus.

Some previous works decomposed the target predictions. Allamanis et al. (2016, 2015a) decomposed method names into smaller "sub-tokens" and used the continuous prediction approach to compose a full name. Iyer et al. (2016) decomposed StackOverflow titles to single words and predicted them word-by-word. In theory, this approach could be used to predict new compositions of names that were not observed in the training corpus, referred to as neologisms (Allamanis et al., 2015a). However, when scaling to millions of examples this approach might become cumbersome and fail to

train well due to hardware and time limitations. As shown in Section 3.5.1, our model yields significantly better results than previous models that used this approach.

Another disadvantage of subtoken-by-subtoken learning is that it requires a time-consuming beam-search during prediction. This results in an *orders-of-magnitude slower prediction rate* (the number of predictions that the model is able to make per second). An empirical comparison of the prediction rate of our model and the models of Allamanis et al. (2016) and Iyer et al. (2016) shows that our model achieves a roughly 200 times faster prediction rate than Iyer et al. (2016) and 10,000 times faster than Allamanis et al. (2016) (Section 3.5.1).

OoV prediction The other possible advantage of Allamanis et al. (2016)'s method — the ability to produce out-of-vocabulary (OoV) predictions by means of a copy mechanism and subtoken-by-subtoken decoding — offer only a negligible contribution. An analysis of our test data shows that the top-10 most frequent method names, such as `toString`, `hashCode` and `equals`, which are typically easy to predict, appear in less than 6% of the test examples. The 13% least occurring names are rare names, which did not appear in their entirety in the training data, and are difficult or impossible to predict exactly even with a neologism or copy mechanism. One example is `imageFormatExceptionShouldProduceNotSuccessOperationResultWithMessage`. However, when trained and evaluated on the same corpus as our model, less than 3% of the predictions of each of these baselines were actually neologisms or OoV. Moreover, in most of the cases where the baseline suggested a neologism or OoV, *it could have produced a more accurate prediction using only already seen target names.*

We thus believe that our efforts would be better spent on the prediction of complete names.

Granularity of path decomposition An alternative approach could decompose the representation of a path to granularity of single nodes and learn to represent a whole path node-by-node using a recurrent neural network (RNN). This would possibly require less space but would also be more time consuming.

Furthermore, a statistical analysis of our corpus shows that more than 95% *of the paths in the test set were already seen in the training set.* Accordingly, in the trade-off between time and space we chose a slightly less expressive, more memory-consuming, but fast-to-train approach. This choice leads to results that are as 95% as good as our final results in only 6 hours of training, while significantly improving over previous works. Despite our choice of time over space, training our model on millions of examples fits in the memory of common GPUs.

3.5 Evaluation

The main contribution of our method is its ability to aggregate an arbitrary sized snippet of code into a fixed-size vector in a way that captures its semantics. Since Java methods are usually short, focused, have a single functionality and a descriptive name, a natural benchmark of our approach would consider a method body as a code snippet, and use the produced code vector to predict the method name. Succeeding in this task would suggest that the code vector has indeed accurately captured the functionality and semantic role of the method.

Our evaluation aims to answer the following questions:

- How useful is our model in predicting method names, and how well does it measure in comparison to other recent approaches (Section 3.5.1)?
- What is the contribution of the attention mechanism to the model? How well would it perform using *hard* attention instead, or using no attention at all (Section 3.5.2)?
- What is the contribution of each of the path-context components to the model (Section 3.5.3)?
- Is it actually able to predict names of complex methods, or only of trivial ones (Section 3.5.4)?
- What are the properties of the learned vectors? Which semantic patterns do they encode (Section 3.5.4)?

Training process In our experiments we took the top $1M$ paths — those that occurred the most in the training set. We used the Adam optimization algorithm (Kingma and Ba, 2014), an adaptive gradient descent method commonly used in deep learning. We used dropout (Srivastava et al., 2014) of 0.25 on the context vectors. The values of all the parameters were initialized using the initialization heuristic of Glorot and Bengio (2010). When training on a single Tesla K80 GPU, we achieved a training throughput of more than 1000 methods per second. Therefore, a single training epoch takes about 3 hours, and it takes about 1.5 days to completely train a model. Training on newer GPUs doubles and quadruples the speed. Although the attention mechanism can aggregate an arbitrary number of inputs, we randomly sampled up to $k = 200$ path-contexts from each training example. The value $k = 200$ seemed to be enough to "cover" each method, since increasing to $k = 300$ did not seem to improve the results.

Datasets We wanted to evaluate the ability of the approach to generalize across projects. We used a dataset of $10,072$ Java GitHub repositories, originally introduced by Alon et al. (2018). Following recent work which found a large amount of code duplication in GitHub (Lopes et al., 2017), Alon et al. (2018) used the top-ranked

Table 3.2: Size of data used in the experimental evaluation.

	Number of methods	Number of files	Size (GB)
Training	12,636,998	1,712,819	30
Validation	371,364	50,000	0.9
Test	368,445	50,000	0.9
Sampled Test	7,454	1,000	0.04

and most popular projects, in which duplication was observed to be less of a problem. Additionally, they filtered out migrated projects and forks of the same project. While it is possible that some duplications are left between the training and test set, in this case the compared baselines could have benefited from them as well. In this dataset, the files from all the projects were shuffled and split to $12,636,998$ training, $371,364$ validation and $368,445$ test methods.

We trained our model on the training set and tuned hyperparameters on the validation set for maximizing F1 score. The number of training epochs was tuned on the validation set using early stopping. Finally, we report results on the unseen test set. A summary of the amount of data used is shown in Table 3.2.

Evaluation metric Ideally, we would have liked to manually evaluate the results, but given that manual evaluation is very difficult to scale, we adopted the measure used in previous works (Allamanis et al., 2016; Alon et al., 2018; Allamanis et al., 2015a), which measured precision, recall, and F1 score over sub-tokens, case-insensitive. This is based on the idea that the quality of a method name prediction depends mainly on the sub-words used to compose it. For example, for a method called `countLines`, a prediction of `linesCount` is considered as an exact match, a prediction of `count` has full precision but low recall, and a prediction of `countBlankLines` has full recall but low precision. An unknown sub-token in the test label ("UNK") is counted as a false negative, therefore automatically hurting recall.

While there are alternative metrics in the literature, such as accuracy and BLEU score, they are problematic because accuracy counts even mostly correct predictions as completely incorrect, and the BLEU score tends to favor short predictions, which are usually uninformative. We provide a qualitative evaluation including a manual inspection of examples in Section 3.5.4.

3.5.1 Quantitative Evaluation

We compare our model to two other recently proposed models that address similar tasks:

CNN+attention — proposed by Allamanis et al. (2016) for prediction of method names using CNNs and attention. This baseline was evaluated on a random sample

Table 3.3: Evaluation comparison between our model and previous works.

	Sampled Test Set			Full Test Set		
Model	Precision	Recall	F1	Precision	Recall	F1
Allamanis et al. (2016)	47.3	29.4	33.9	-	-	-
Iyer et al. (2016)	27.5	21.5	24.1	33.7	22.0	26.6
Alon et al. (2018)	-	-	-	53.6	46.6	49.9
code2vec (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**

of the test set due to its slow prediction rate (Table 3.3). We note that the F1 score reported here is lower than the original results reported in their paper, because we consider the task of learning *a single model that is able to predict names for a method from any possible project*. We do not make the restrictive assumption of having a per-project model, able to predict only names within that project. The results we report for CNN+attention are when evaluating their technique in this realistic setting. In contrast, the numbers reported in their original work are for the simplified setting of predicting names *within the scope of a single project*.

LSTM+attention — proposed by Iyer et al. (2016), originally for translation between StackOverflow questions in English and snippets of code that were posted as answers and vice versa, using an encoder-decoder architecture based on LSTMs and attention. Originally, they demonstrated their approach for C# and SQL. We used a Java lexer instead of the original C#, and carefully modified it to be equivalent. We re-trained their model with the target language being the methods' names, split into sub-tokens. Note that this model was designed for a slightly different task than ours: translation between source code snippets and natural language descriptions, and not specifically for prediction of method names.

Paths+CRFs — proposed by Alon et al. (2018), using a similar syntactic path representation as this work, with CRFs as the learning algorithm. We evaluate our model on their introduced dataset, and achieve a significant improvement in results, training time and prediction time.

Each baseline was trained on the same training data as our model. We used their default hyperparameters, except for the embedding and LSTM size of the LSTM+attention model, which were reduced from 400 to 100, to allow it to scale to our enormous training set while complying with the GPU's memory constraints. The alternative was to reduce the amount of training data, which achieved worse results.

Performance Table 3.3 shows the precision, recall, and F1 score of each model. The model of Alon et al. (2018) seems to perform better than that of Allamanis et al. (2016) and Iyer et al. (2016), while our model achieves significantly better precision and recall than all of them.

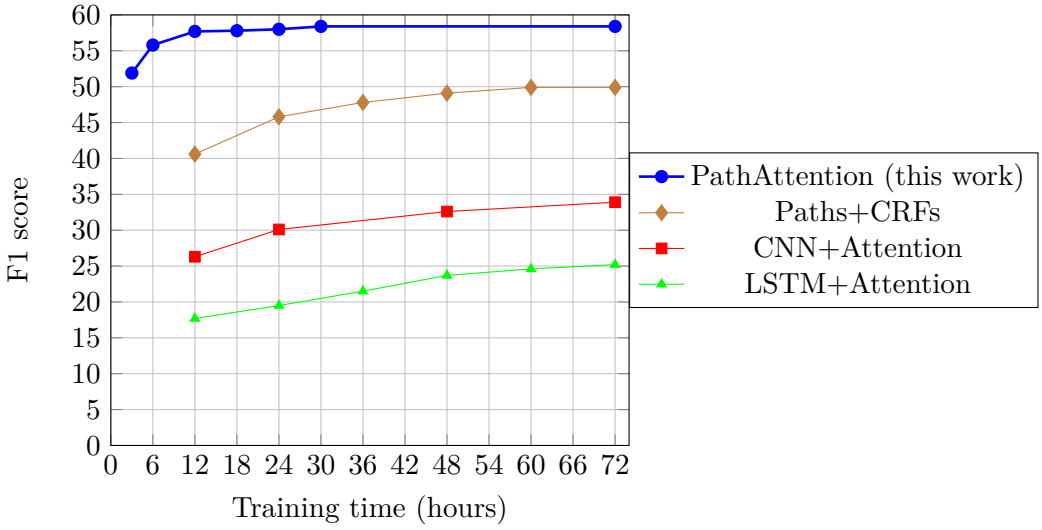


Figure 3.5: Our model achieves significantly better results than the baselines and in shorter time.

Short and long methods The reported results are based on evaluation on *all* the test data. Additionally evaluating the performance of our model with respect to the length of a test method, we observe similar results across method lengths, with a natural decrease in performance as the length increases. For example, the F1 score of one-line methods is around 65; for two-to-ten lines 59; and for eleven-lines and more 52, while the average method length is 7 lines. We used all the methods in the dataset, regardless of their size. This shows the robustness of our model to the length of the methods. Short methods have shorter names and their logic is usually simpler, while long methods benefit from more context for prediction, but their names are usually longer, more diverse and sparse, for example: `generateTreeSetHashSetSpoofingSetInteger`, which has 17 lines of code.

Speed Figure 3.5 shows the test F1 score over training time for each of the evaluated models. In just 3 hours, our model achieves results that are as 88% as good as its final results, and in 6 hours results that are as 95% as good, with both being substantially higher than the best results of the baseline models. Our model achieves its best results after 30 hours.

Table 3.3 shows the approximate prediction rate of the different models. The syntactic preprocessing time of our model is negligible but is included in the calculation. As shown, due to their complexity and expensive beam search on prediction, the other models are several orders of magnitude slower than ours, limiting their applicability.

Data efficiency The results reported in Table 3.3 were obtained using our full and large training corpus, to demonstrate the ability of our approach to leverage enormous amounts of training data in a relatively short training time. However, in order to in-

Table 3.4: Comparison of model designs.

Model Design	Precision	Recall	F1
No-attention	54.4	45.3	49.4
Hard attention	42.1	35.4	38.5
Train-soft, predict-hard	52.7	45.9	49.1
Soft attention	**63.1**	**54.4**	**58.4**
Element-wise soft attention	**63.7**	**55.4**	**59.3**

vestigate the data efficiency of our model, we also performed experiments using smaller training corpora which are not reported in detail here. With 20% of the data, the F1 score of our model drops to only 50%. With 5% of the data, the F1 score drops only to 30% of our top results. We do not focus on this series of experiments here: since our model can process more than a thousand of examples per second, there is no real reason to deliberately limit the size of the training corpus.

3.5.2 Evaluation of Alternative Designs

We experiment with alternative model designs, in order to understand the contribution of each network component.

Attention As we refer to our approach as *soft attention*, we examine two other approaches at the opposite extreme:

1. *No-attention* — in which every path-context is given an *equal* weight: the model uses the ordinary average of the path-contexts rather than learning a weighted average.

2. *Hard attention* — in which instead of focusing the attention "softly" over the path-contexts, all the attention is given to a single path-context, i.e., the network learns to select a *single* most important path-context at a time.

A new model was trained for each of these alternative designs. However, training hard-attention neural networks is difficult, because the gradient of the $argmax$ function is zero almost everywhere. Therefore, we experimented with an additional approach: *train-soft, predict-hard*, in which training is performed using soft attention (as in our ordinary model), and prediction is performed using hard attention. Table 3.4 shows the results of all the compared alternative designs. As seen, hard attention achieves the lowest results. This concludes that when predicting method names, or in general describing code snippets, it is more beneficial to use all the contexts with equal weights than to focus on the single most important one. *Train-soft, predict-hard* improves over hard training, and gains similar results to no-attention. As soft attention achieves higher scores than all of the alternatives, both on training and prediction, this

Table 3.5: Our model while hiding input components.

Path-context input		Precision	Recall	F1
Full:	$\langle x_s, p, x_t \rangle$	63.1	54.4	58.4
Only-values:	$\langle x_s, \_\_\_, x_t \rangle$	44.9	37.1	40.6
No-values:	$\langle \_\_\_, p, \_\_\_ \rangle$	12.0	12.6	12.3
Value-path:	$\langle x_s, p, \_\_\_ \rangle$	31.5	30.1	30.7
One-value:	$\langle x_s, \_\_\_, \_\_\_ \rangle$	10.6	10.4	10.7

experiment shows its contribution as a "sweet-spot" between no-attention and hard attention.

Removing the fully-connected layer To understand the contribution of each component of our model, we experiment with removing the fully connected layer (described in Section 3.4.2). In this experiment, soft attention is applied directly on the *context-vectors* instead of the *combined context-vectors*. This experiment resulted in the same final F1 score as our regular model. Even though its training rate (training examples per second) was faster, it took more actual training time to achieve the same results. For example, it took 12 hours instead of 6 to reach results that are as 95% as good as the final results, and a few more hours to achieve the final results.

Element-wise soft attention We also experimented with *element-wise soft attention*. In this design, instead of using a single attention vector $\boldsymbol{a} \in \mathbb{R}^d$ to compute the attention for the whole combined context vector $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$, there are d attention vectors $\boldsymbol{a_1}, ..., \boldsymbol{a_d} \in \mathbb{R}^d$, and each of them is used to compute the attention for a different *element*. Therefore, the attention weight for element j of a combined context vector $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ is: attention weight $\alpha_{i_j} = \frac{\exp(\tilde{\boldsymbol{c}}_{\boldsymbol{i}}^T \cdot \boldsymbol{a_j})}{\sum_{k=1}^{n} \exp(\tilde{\boldsymbol{c}}_{\boldsymbol{k}}^T \cdot \boldsymbol{a_j})}$. This variation allows the model to compute a different attention score for each *element* in the combined context vector, instead of computing the same attention score for the whole combined context vector. This model achieved an F1 score of 59.3 (on the full test set), which is even higher than our standard soft attention model, but since this model gives different attention to different elements within the same context vector, it is more difficult to interpret. Thus, this is an alternative model that gives slightly better results at the cost of poor interpretability and slower training.

3.5.3 Data Ablation Study

The contribution of each path-context element To understand the contribution of each component of a path-context, we evaluate our best model on the same test set in the same settings, except that one or more input locations is "hidden" and replaced with a constant "UNK" symbol, such that the model cannot use this element for prediction. As the "full" representation is referred to as: $\langle x_s, p, x_t \rangle$, the following experiments were

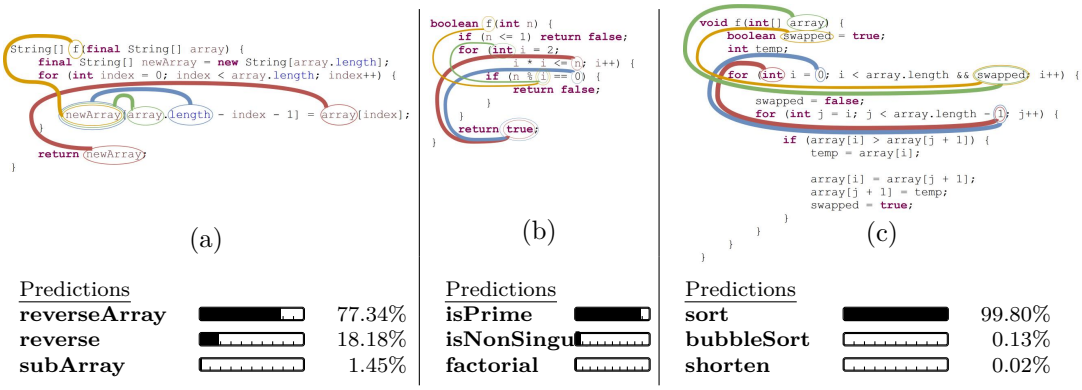


Figure 3.6: Example predictions from our model, with the top-4 paths that were given the most attention for each code snippet. The width of each path is proportional to the attention it was given by the model.

performed:

- "only-values" - using only the values of the terminals for prediction, without paths, and therefore representing each path-context as: $\langle x_s, \_\_\_, x_t \rangle$.
- "no-values" - using only the path: $\langle \_\_\_, p, \_\_\_ \rangle$, without identifiers and keywords.
- "value-path" - allowing the model to use a path and one of its values: $\langle x_s, p, \_\_\_ \rangle$.
- "one-value" - using only one of the values: $\langle x_s, \_\_\_, \_\_\_ \rangle$.

The results of these experiments are presented in Table 3.5. Interestingly, the "full" representation ($\langle x_s, p, x_t \rangle$) achieves better results than the sum of "only-values" and "no-values", without each of them alone "covering" for the other. This shows the importance of using *both* paths and keywords, and letting the attention mechanism learn how to combine them in every example. The poorer results of "only-values" (compared to the full representation) show the importance of using syntactic paths. As shown in the table, dropping identifiers and keywords hurt the model more than dropping paths, but combining them achieves significantly better results. Better results are obtained for "no-paths" than for "no-values", and "single-identifiers" obtains the worst results.

The poor results of "no-values" suggest that predicting names for methods with obfuscated names is a much more difficult task. In this scenario, it might be more beneficial to predict variable names as a first step using a model that was trained specifically for this task, and then predict a method name given the predicted variable names.

3.5.4 Qualitative Evaluation

Interpreting Attention

Despite the "black-box" reputation of neural networks, our model is partially interpretable thanks to the attention mechanism, which allows us to visualize the distribution of weights over the bag of path-contexts. Figure 3.6 illustrates a few predictions, along with the path-contexts that were given the most attention in each method. The width of each of the visualized paths is proportional to the attention weight that it was allocated. We note that in these figures the path is represented only as a connecting line between tokens, while in fact it contains rich syntactic information which is not expressed properly in the figures. Figure 3.7 and Figure 3.8 portray the paths on the AST.

The examples of Figure 3.6 are particularly interesting since the top names are accurate and descriptive (`reverseArray` and `reverse`; `isPrime`; `sort` and `bubbleSort`) but do not appear explicitly in the method bodies. The method bodies, and specifically the path-contexts that were given the most attention, describe lower-level operations. Suggesting a descriptive name for each of these methods is difficult and might take time even for a trained human programmer. The average method length in our dataset of real-world projects is 7 lines, and the examples presented in this section are longer than this average length.

Figure 3.7 and Figure 3.8 show additional predictions of our model, along with the path-contexts that were given the most attention in each example. The path-contexts are portrayed both on the code and on the AST. An interactive demo of method name predictions and name vector similarities can be found at: `http://code2vec.org`. When manually examining the predictions of custom inputs, it is important to note that a machine learning model learns to predict names for examples that are likely to be observed "in the wild". Thus, it can be misled by confusing adversarial examples that are unlikely to be found in real code.

Semantic Properties of the Learned Embeddings

Surprisingly, the learned method name vectors encode many semantic similarities and even analogies that can be represented as linear additions and subtractions. When simply looking for the closest vector (in terms of cosine distance) to a given method name vector, the resulting neighbors usually contain semantically similar names; e.g. `size` is most similar to `getSize`, `length`, `getCount`, and `getLength`. Table 3.1 shows additional examples of name similarities.

When looking for a vector that is close to *two* other vectors, we often find names that are semantic combinations of the two other names. Specifically, we can look for the vector $\boldsymbol{v}$ that maximizes the similarity to two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$:

$$argmax_{\boldsymbol{v}\in V}\left(sim\left(\boldsymbol{a},\boldsymbol{v}\right)\circledast sim\left(\boldsymbol{b},\boldsymbol{v}\right)\right) \tag{3.3}$$

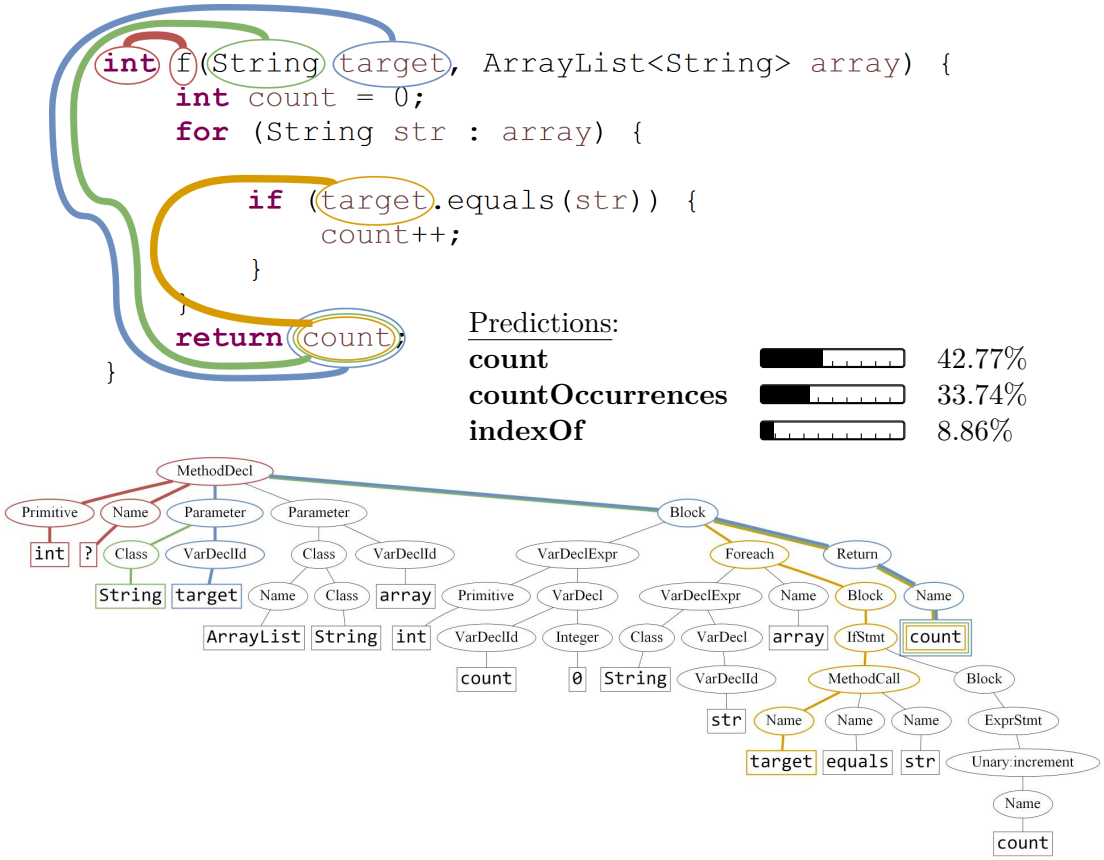


Figure 3.7: An example for a method name prediction, portrayed on the AST. The top-four path-contexts were given a similar attention, which is higher than the rest of the path-contexts.

Table 3.6: Semantic combinations of method names.

A	+B	≈C
get	value	getValue
get	instance	getInstance
getRequest	addBody	postRequest
setHeaders	setRequestBody	createHttpPost
remove	add	update
decode	fromBytes	deserialize
encode	toBytes	serialize
equals	toLowerCase	equalsIgnoreCase

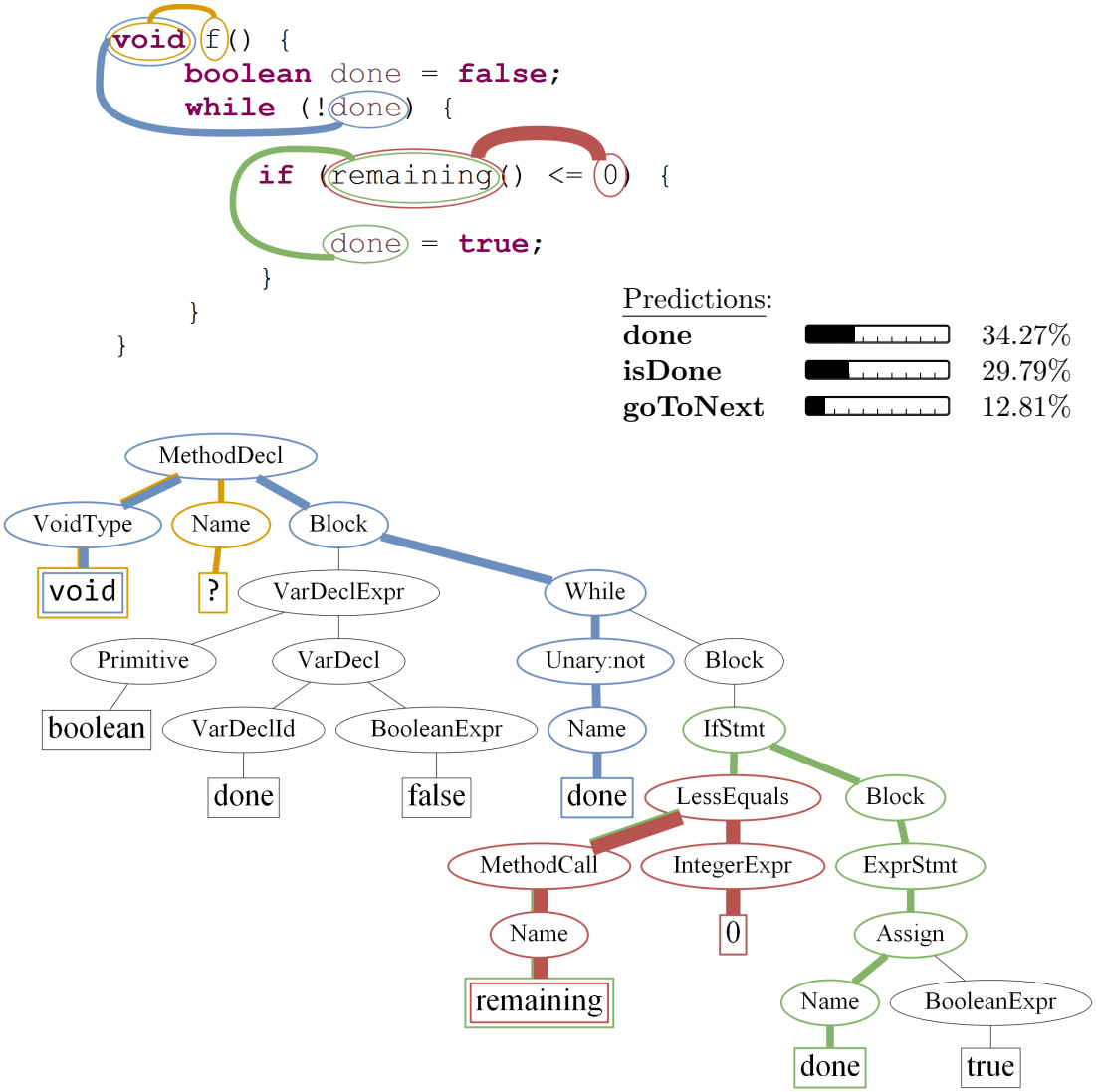


Figure 3.8: An example for a method name prediction, portrayed on the AST. The width of each path is proportional to the attention it was given.

Table 3.7: Semantic analogies between method names.

A :	B	C :	D
open :	connect	close :	disconnect
key :	keys	value :	values
lower :	toLowerCase	upper :	toUpperCase
down :	onMouseDown	up :	onMouseUp
warning :	getWarningCount	error :	getErrorCount
value :	containsValue	key :	containsKey
start :	activate	end :	deactivate
receive :	download	send :	upload

where $\circledast$ is an arithmetic operator used to combine two similarities, and V is a vocabulary of learned name vectors, *tags_vocab* in our case. When measuring similarity using cosine distance, Equation (3.3) can be written as:

$$argmax_{\boldsymbol{v} \in V} \left(cos\left(\boldsymbol{a}, \boldsymbol{v}\right) \circledast cos\left(\boldsymbol{b}, \boldsymbol{v}\right) \right) \tag{3.4}$$

Neither *vec*(`equals`) nor *vec*(`toLowerCase`) are the closest vectors to *vec*(`equalsIgnoreCase`) individually. However, assigning $\boldsymbol{a} = vec\left(\texttt{equals}\right)$, $\boldsymbol{b} = vec\left(\texttt{toLowerCase}\right)$ and using "+" as the operator $\circledast$, results in the vector of `equalsIgnoreCase` as the vector that maximizes Equation (3.4) for $\boldsymbol{v}$.

Previous work in NLP has suggested a variety of methods for combining similarities (Levy and Goldberg, 2014b) for the task of natural language analogy recovery. Specifically, when using "+" as the operator $\circledast$, as done by Mikolov et al. (2013b), and denoting $\hat{\boldsymbol{u}}$ as the unit vector of a vector $\boldsymbol{u}$, Equation (3.4) can be simplified to:

$$argmax_{\boldsymbol{v} \in V} \left(\hat{\boldsymbol{a}} + \hat{\boldsymbol{b}} \right) \cdot \hat{\boldsymbol{v}}$$

since cosine distance between two vectors equals the dot product of their unit vectors. This provides us with a simpler method for finding the above combination of method name similarities:

$$vec\left(\texttt{equals}\right) + vec\left(\texttt{toLowerCase}\right) \approx vec\left(\texttt{equalsIgnoreCase}\right)$$

This implies that the model has learned that `equalsIgnoreCase` is the most similar name to `equals` *and* `toLowerCase` combined. Table 3.6 shows some of these examples.

Just as Mikolov et al. (2013a,c) used vector calculation to express syntactic and semantic word analogies in NLP, the method name vectors learned by our model also express similar syntactic and semantic analogies. For example, *vec*(`download`)-*vec*(`receive`)+*vec*(`send`) results in a vector whose closest neighbor is the vector for `upload`. This analogy can be read as: "`receive` is to `send` as `download` is to: `upload`". More examples are shown in Table 3.7.

3.6 Limitations of the code2vec model

In this section we discuss some limitations of our model and potential future research directions. Some of these limitations will be solved by CODE2SEQ, which will be presented in the next chapter.

Closed labels vocabulary One of the major limiting factors is the closed label space we use as target: our model is able to predict only labels that were observed as is at training time. This works very well for the vast majority of targets (that repeat across multiple programs), but as the targets become very specific and diverse (e.g.,

`findUserInfoByUserIdAndKey`) the model is unable to compose such names and usually catches only the main idea (for example: `findUserInfo`). Overall, on a general dataset, our model outperforms the baselines by a large margin even though the baselines are technically able to produce complex names. CODE2SEQ solves this limitation by predicting a *sequence* of output symbols. This exact sequence was not necessarily seen at training time.

Sparsity and Data-hunger There are three main sources of sparsity in our model:

- Terminal values are represented as whole symbols - e.g., each `newArray` and `oldArray` is a unique symbol that has an embedding of its own, even though they share most of their characters (`Array`).
- AST paths are represented as monolithic symbols - two paths that share most of their AST nodes but differ in only a single node are represented as distinct paths which are assigned distinct embeddings.
- Target nodes are whole symbols, even if they are composed of more common smaller symbols.

This sparsity results in the model consuming a lot of trained parameters to keep an embedding for each observed value. The large number of trained parameters results in large GPU memory consumption at training time, increases the size of the stored model (about 1.4 GB), and requires a lot of training data. Furthermore, modeling source code with a finer granularity of atomic units may allow the model to represent more unseen contexts as compositions of smaller atomic units, thus repeating more atomic units across examples. In the model described in this work, paths, terminal values or target values that were not observed during training cannot be represented. To address these limitations we train the model on a huge dataset of more than 12M examples, but the model might not perform as well using smaller datasets. Although requiring a lot of GPU memory, training our model on millions of examples fits in the memory of a relatively old Tesla K80 GPU.

An alternative approach for reducing the sparsity of AST paths is to use *path abstractions* where only parts of the path are used in the context (abstracting away certain kinds of nodes, merging certain kinds of nodes, etc.).

In CODE2SEQ, we solve all these sources of sparsity by decomposing terminals, paths, and targets to smaller components. This significantly improves the results of the CODE2SEQ model on both small and large datasets, while reducing the size of the stored model *by about 90%*.

Dependency on variable names Since we trained our model on top-starred open-source projects where variable naming is usually good, the model has learned to leverage variable names to predict the target label. When given uninformative, obfuscated or

adversarial variable names, the prediction of the label is usually less accurate. We are considering several approaches to address this limitation in future research. One potential solution is to train the model on a mixed dataset of good and hidden variable names, hopefully reducing model dependency on variable names; another solution is to apply a model that was trained for variable de-obfuscation first (such as (Alon et al., 2018; Raychev et al., 2015)) and feed the predicted variable names into our model.

Chapter 4

code2seq: Generating Sequences from Structured Representations of Code

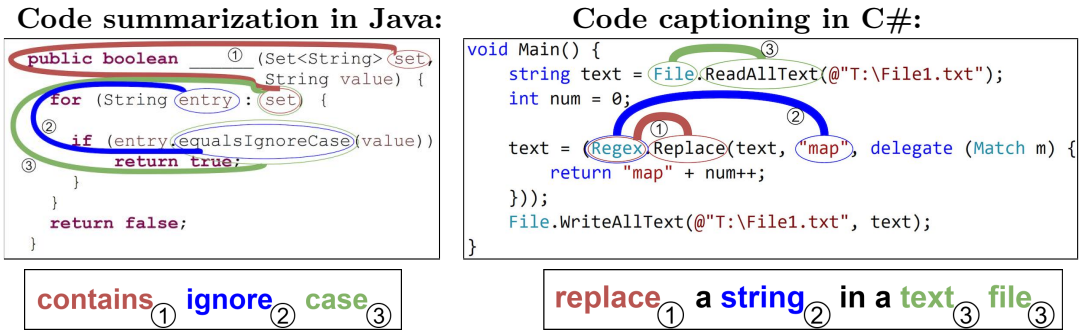
4.1 Introduction

Modeling the relation between source code and natural language can be used for automatic code summarization (Allamanis et al., 2016), documentation (Iyer et al., 2016), retrieval (Allamanis et al., 2015b), and even generation (Balog et al., 2017; Rabinovich et al., 2017; Yin and Neubig, 2017; Devlin et al., 2017; Murali et al., 2018; Brockschmidt et al., 2019). In this work, we consider the general problem of generating a natural language sequence from a given snippet of source code.

A direct approach is to frame the problem as a machine translation problem, where the source sentence is the sequence of tokens in the code and the target sentence is a corresponding natural language sequence. This approach allows one to apply state-of-the-art neural machine translation (NMT) models from the sequence-to-sequence (seq2seq) paradigm (Sutskever et al., 2014; Cho et al., 2014b; Bahdanau et al., 2014; Luong et al., 2015; Vaswani et al., 2017), yielding state-of-the-art performance on various code captioning and documentation benchmarks (Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017) despite having extremely long source sequences.

We present an alternative approach for encoding source code that leverages the syntactic structure of programming languages: CODE2SEQ. We represent a given code snippet as a set of compositional paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector using LSTMs (Hochreiter and Schmidhuber, 1997). During decoding, CODE2SEQ attends over a different weighted average of the path-vectors to produce each output token, much like NMT models attend over token representations in the source sentence.

We show the effectiveness of our code2seq model on two tasks: (1) code summariza-



(a) (b)

Figure 4.1: Example of (a) code summarization of a Java code snippet, and (b) code captioning of a C# code snippet, along with the predictions produced by our models. The highlighted paths in each example are the top-attended paths in each decoding step. Because of space limitations we included only the top-attended path for each decoding step, but hundreds of paths are attended at each step.

tion (Figure 4.1a), where we predict a Java method's name given its body, and (2) code captioning (Figure 4.1b), where we predict a natural language sentence that describes a given C# snippet.

On both tasks, our CODE2SEQ model outperforms models that were explicitly designed for code, such as the model of Allamanis et al. (2016) and CodeNN (Iyer et al., 2016), as well as TreeLSTMs (Tai et al., 2015) and state-of-the-art NMT models (Luong et al., 2015; Vaswani et al., 2017). To examine the importance of each component of the model, we conduct a thorough ablation study. In particular, we show the importance of structural encoding of code, by showing how our model yields a significant improvement over an ablation that uses only token-level information without syntactic paths. To the best of our knowledge, this is the first work to directly use paths in the abstract syntax tree for end-to-end generation of sequences.

4.2 Representing Code as AST Paths

An Abstract Syntax Tree (AST) uniquely represents a source code snippet in a given language and grammar. The leaves of the tree are called *terminals*, and usually refer to user-defined values which represent identifiers and names from the code. The non-leaf nodes are called *nonterminals* and represent a restricted set of structures in the language, e.g., loops, expressions, and variable declarations. For example, Figure 4.2c shows a partial AST for the code snippet of Figure 4.2a. Names (such as `num`) and types (such as `int`) are represented as values of terminals; syntactic structures such as variable declaration (`VarDec`) and a do-while loop (`DoStmt`) are represented as nonterminals.

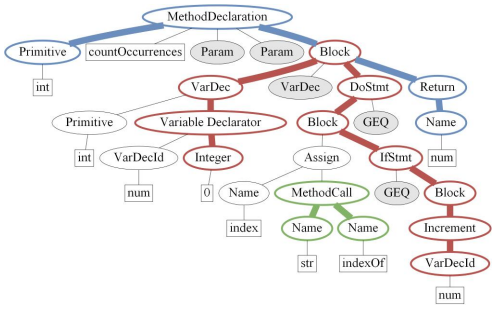
Given the AST of a code snippet, we consider all pairwise paths between terminals, and represent them as sequences of terminal and nonterminal nodes. We then use these paths with their terminals' values to represent the code snippet itself. For example,

```java
int countOccurrences(String str,
                     char ch) {
  int num = 0;
  int index = -1;
  do {
    index = str.indexOf(ch, index + 1);
    if (index >= 0) {
      num++;
    }
  } while (index >= 0);
  return num;
}
```

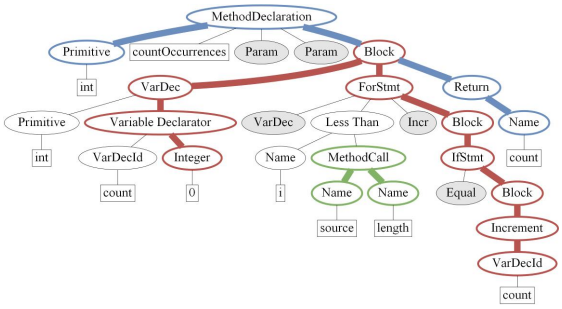
(a)

```java
int countOccurrences(String source,
                     char value) {
  int count = 0;
  for (int i = 0; i < source.length(); i++) {
    if (source.charAt(i) == value) {
      count++;
    }
  }
  return count;
}
```

(b)



(c)



(d)

Figure 4.2: An example of two Java methods that have exactly the same functionality. Although these methods have different *sequential* (token-based) representations, repeating paths, which might differ in only a single node (a `ForStmt` node instead of a `Do`-while node), will be revealed if we consider syntactic patterns.

consider the two Java methods of Figure 4.2. Both of these methods count occurrences of a character in a string. They have exactly the same functionality, although a different implementation, and therefore different surface forms. If these snippets are encoded as sequences of tokens, the recurring patterns that suggest the common method name might be overlooked. However, a structural observation reveals syntactic paths that are common to both methods, and differ only in a single node of a `Do`-while statement versus a `For` statement. This example shows the effectiveness of a syntactic encoding of code. Such an encoder can generalize much better to unseen examples because the AST normalizes a lot of the surface form variance. Since our encoding is compositional, the encoder can generalize even if the paths are not identical (e.g., a `For` node in one path and a `While` in the other).

Since a code snippet can contain an arbitrary number of such paths, we sample k paths as the representation of the code snippet. To avoid bias, k new paths are sampled afresh in every training iteration. In Section 4.5 we show that this runtime-sampling provides regularization and improves results compared to sampling the same k paths for each example in advance.

Formally, we use $\mathcal{C}$ to denote a given snippet of code. Every training iteration, k pairs of terminals are uniformly sampled from within the AST of $\mathcal{C}$. Each pair of terminals $\left(v_1^i, v_{l_i}^i\right)$ implies a single path between them: $v_1^i v_2^i ... v_{l_i}^i$. Finally, the input code example is represented as a set of these k random AST paths: $\left\{\left(v_1^1 v_2^1 ... v_{l_1}^1\right), ..., \left(v_1^k v_2^k ... v_{l_k}^k\right)\right\}$, where l_j is the length of the jth path.

4.3 Model Architecture

Our model follows the standard encoder-decoder architecture for NMT (Section 4.3.1), with the significant difference that *the encoder does not read the input as a flat sequence of tokens*. Instead, the encoder creates a vector representation for each AST path separately (Section 4.3.2). The decoder then attends over the encoded AST paths (rather than the encoded tokens) while generating the target sequence. Our model is illustrated in Figure 4.3.

4.3.1 Encoder-Decoder Framework

Contemporary NMT models are largely based on an encoder-decoder architecture (Cho et al., 2014b; Sutskever et al., 2014; Luong et al., 2015; Bahdanau et al., 2014), where the encoder maps an input sequence of tokens $\boldsymbol{x} = (x_1, ..., x_n)$ to a sequence of continuous representations $\boldsymbol{z} = (z_1, ..., z_n)$. Given $\boldsymbol{z}$, the decoder then generates a sequence of output tokens $\boldsymbol{y} = (y_1, ..., y_m)$ one token at a time, hence modeling the conditional probability: $p\left(y_1, ..., y_m | x_1, ..., x_n\right)$.

At each decoding step, the probability of the next target token depends on the previously generated token, and can therefore be factorized as:

$$p\left(y_1, ..., y_m | x_1, ..., x_n\right) = \prod_{j=1}^{m} p\left(y_j | y_{<j}, z_1, ..., z_n\right)$$

In attention-based models, at each time step t in the decoding phase, a context vector c_t is computed by attending over the elements in $\boldsymbol{z}$ using the decoding state h_t, typically computed by an LSTM.

$$\boldsymbol{\alpha}^t = softmax\left(h_t W_a \boldsymbol{z}\right) \qquad c_t = \sum_{i}^{n} \alpha_i^t z_i$$

The context vector c_t and the decoding state h_t are then combined to predict the current target token y_t. Previous work differs in the way the context vector is computed and in the way it is combined with the current decoding state. A standard approach (Luong et al., 2015) is to pass c_t and h_t through a multi-layer perceptron (MLP) and then predict the probability of the next token using softmax:

$$p\left(y_t | y_{<t}, z_1, ..., z_n\right) = softmax\left(W_s tanh\left(W_c\left[c_t; h_t\right]\right)\right)$$

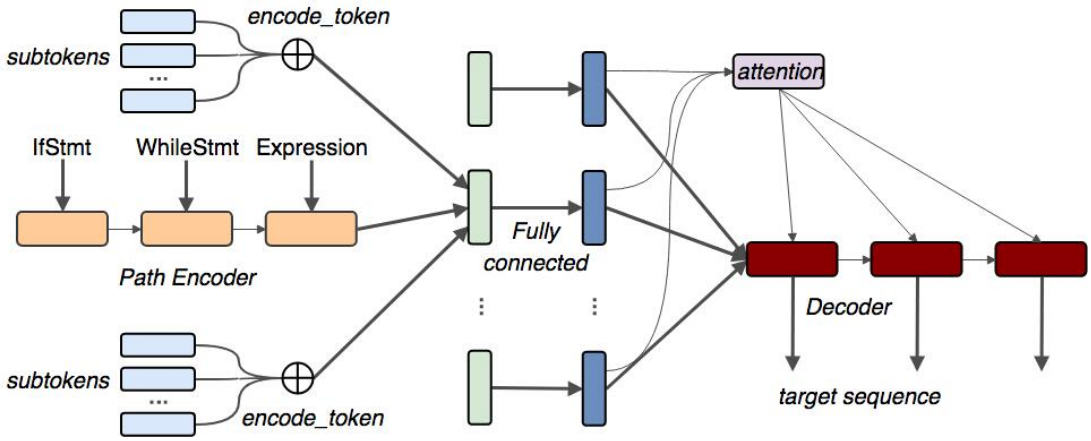


Figure 4.3: Our model encodes each AST path with its values as a vector, and uses the average of all of the k paths as the decoder's start state. The decoder generates an output sequence while attending over the k encoded paths.

4.3.2 AST Encoder

Given a set of AST paths $\{x_1, ..., x_k\}$, our goal is to create a vector representation z_i for each path $x_i = v_1^i v_2^i ... v_{l_i}^i$. We represent each path separately using a bi-directional LSTM to encode the path, and sub-token embeddings to capture the compositional nature of the terminals' values (the tokens).

Path Representation Each AST path is composed of nodes and their child indices from a limited vocabulary of up to 364 symbols. We represent each node using a learned embedding matrix E^{nodes} and then encode the entire sequence using the final states of a bi-directional LSTM:

$$h_1, ..., h_l = LSTM(E_{v_1}^{nodes}, ..., E_{v_l}^{nodes})$$

$$encode\_path(v_1 ... v_l) = [h_l^{\rightarrow}; h_1^{\leftarrow}]$$

Token Representation The first and last node of an AST path are terminals whose values are tokens in the code. Following Allamanis et al. (2015a, 2016), we split code tokens into *sub*tokens; for example, a token with the value `ArrayList` will be decomposed into `Array` and `List`. This is somewhat analogous to byte-pair encoding in NMT (Sennrich et al., 2016), although in the case of programming languages, coding conventions such as camel notation provide us with an explicit partition of each token. We use a learned embedding matrix $E^{subtokens}$ to represent each subtoken, and then sum the subtoken vectors to represent the full token:

$$encode\_token(w) = \sum_{s \in split(w)} E_s^{subtokens}$$

The LSTM decoder may also predict subtokens at each step (e.g. when generating method names), although the decoder's subtoken embedding matrix will be different.

Combined Representation To represent the path $x = v_1...v_l$, we concatenate the path's representation with the token representations of each terminal node, and apply a fully-connected layer:

$$z = tanh\left(W_{in}\left[encode\_path(v_1...v_l); encode\_token(value(v_1)); encode\_token(value(v_l))\right]\right)$$

where *value* is the mapping of a terminal node to its associated value, and W_{in} is a $(2d_{path} + 2d_{token}) \times d_{hidden}$ matrix.

Decoder Start State To provide the decoder with an initial state, we average the combined representations of *all* the k paths in the given example:

$$h_0 = \frac{1}{k}\sum_{i=1}^{k} z_i$$

Unlike typical encoder-decoder models, the order of the input random paths is not taken into account. Each path is encoded separately and the combined representations are aggregated with mean pooling to initialize the decoder's state. This represents the given source code as a *set* of random paths.

Attention Finally, the decoder generates the output sequence while *attending over all of the combined representations* $z_1, ...z_k$, similarly to the way that seq2seq models attend over the source symbols. The attention mechanism is used to dynamically select the distribution over these k combined representations while decoding, just as an NMT model would attend over the encoded source tokens.

4.4 Experiments

We evaluate our model on two code-to-sequence tasks: summarization (Section 4.4.1), in which we predict Java methods' names from their bodies, and captioning (Section 4.4.2), where we generate natural language descriptions of C# code snippets. Although out of the focus of this work, in Section 4.4.3 we show that our model also generates Javadocs more accurately than an existing work. We thus demonstrate that our approach can produce both method names and natural language outputs, and can encode a code snippet in any language for which an AST can be constructed (i.e., a parser exists).

Setup The values of all of the parameters are initialized using the initialization heuristic of Glorot and Bengio (2010). We optimize the cross-entropy loss (Rubinstein, 1999,

2001) with a Nesterov momentum (Nesterov, 1983) of 0.95 and an initial learning rate of 0.01, decayed by a factor of 0.95 every epoch. For the Code Summarization task, we apply dropout (Srivastava et al., 2014) of 0.25 on the input vectors x_j, and 0.7 for the Code Captioning task because of the smaller number of examples in the C# dataset. We apply a recurrent dropout of 0.5 on the LSTM that encodes the AST paths. We used $d_{tokens} = d_{nodes} = d_{hidden} = d_{target} = 128$. For the Code Summarization task, each LSTM that encodes the AST paths had 128 units and the decoder LSTM had 320 units. For the Code Captioning task, to support the longer target sequences, each encoder LSTM had 256 units and the decoder was of size 512.

Choice of k We experimented with different values of k, the number of sampled paths from each example (which we set to 200 in the final models). Lower values than $k = 100$ showed worse results, and increasing to $k > 300$ did not result in consistent improvement. In practice, $k = 200$ was found to be a reasonable sweet spot between capturing enough information while keeping training feasible in the GPU's memory. Additionally, since the average number of paths in our Java-large training set is 220 paths per example, a number as high as 200 is beneficial for some large methods.

4.4.1 Code Summarization

In this task, we predict a Java method's name given its body. As was previously observed (Allamanis et al., 2016; Alon et al., 2019c), this is a good benchmark because a method name in open-source Java projects tends to be succinct and precise, and a method body is often a complete logical unit. We predict the target method name as a sequence of sub-tokens, e.g., `setMaxConnectionsPerServer` is predicted as the sequence "set max connections per server". The target sequence length is about 3 on average. We adopt the measure used by Allamanis et al. (2016) and Alon et al. (2019c), who measured precision, recall, and F1 score over the target sequence, case insensitive.

Data We experiment with this task across three datsets. In these datasets, we always train across multiple projects and predict on distinct projects:

Java-small – Contains 11 relatively large Java projects, originally used for 11 distinct models for training and predicting within the scope of the same project (Allamanis et al., 2016). We use the same data, but *train and predict across projects*: we took 9 projects for training, 1 project for validation and 1 project as our test set. This dataset contains about $700K$ examples.

Java-med – A new dataset of the 1000 top-starred Java projects from GitHub. We randomly select 800 projects for training, 100 for validation and 100 for testing. This dataset contains about 4M examples and we make it publicly available.

Java-large – A new dataset of the 9500 top-starred Java projects from GitHub that were created since January 2007. We randomly select 9000 projects for training, 250

for validation and 300 for testing. This dataset contains about $16M$ examples and we make it publicly available.

Baselines We re-trained all of the baselines on all of the datasets of this task using the original implementations of the authors. We compare CODE2SEQ to the following baselines: Allamanis et al. (2016), who used a convolutional attention network to predict method names; syntactic paths with Conditional Random Fields (CRFs) (Alon et al., 2018); code2vec (Alon et al., 2019c); and a TreeLSTM (Tai et al., 2015) encoder with an LSTM decoder and attention on the input sub-trees. Additionally, we compared to three NMT baselines that read the input source code as a stream of tokens: 2-layer bidirectional encoder-decoder LSTMs (split tokens and full tokens) with global attention (Luong et al., 2015), and the Transformer (Vaswani et al., 2017), which achieved state-of-the-art results for translation tasks.

We put significant effort into strengthening the NMT baselines in order to provide a fair comparison: (1) we split tokens to subtokens, as in our model (e.g., `HashSet` → `Hash Set`) – this was shown to improve the results by about 10 F1 points (Figure 4.4); (2) we deliberately kept the original casing of the source tokens since we found it to improve their results; and (3) during inference, we replaced generated UNK tokens with the source tokens that were given the highest attention. For the 2-layer BiLSTM we used embeddings of size 512, an encoder and a decoder of 512 units each, and the default hyperparameters of OpenNMT (Klein et al., 2017). For the Transformer, we used their original hyperparameters (Vaswani et al., 2017). This resulted in a Transformer model with $169M$ parameters and a BiLSTM model with $134M$ parameters, while our code2seq model had only 37M.[1]

Performance Figure 4.4 shows the results for the code summarization task. Our model significantly outperforms the baselines in both precision and recall across all three datasets, demonstrating that there is added value in leveraging ASTs to encode source code. Our model improves over the best baselines, BiLSTM with split tokens, by between 4 to 8 F1 points on all benchmarks. BiLSTM with split tokens consistently scored about 10 F1 points more than BiLSTM with full tokens, and for this reason we included only the split token Transformer and TreeLSTM baselines. Our model outperforms ConvAttention (Allamanis et al., 2016), which was designed specifically for this task; Paths+CRFs (Alon et al., 2018), which used syntactic features; and TreeLSTMs. Although TreeLSTMs also leverage syntax, we hypothesize that our syntactic paths capture long distance relationships while TreeLSTMs capture mostly local properties. Examples for predictions made by our model and each of the baselines can be found in Alon et al. (2019a) and at `http://code2seq.org`.

[1]We also trained versions of the NMT baselines in which we down-matched the sizes and number of parameters to our model. These baselines seemed to benefit from more parameters, so the results reported here are for the versions that had many more parameters than our model.

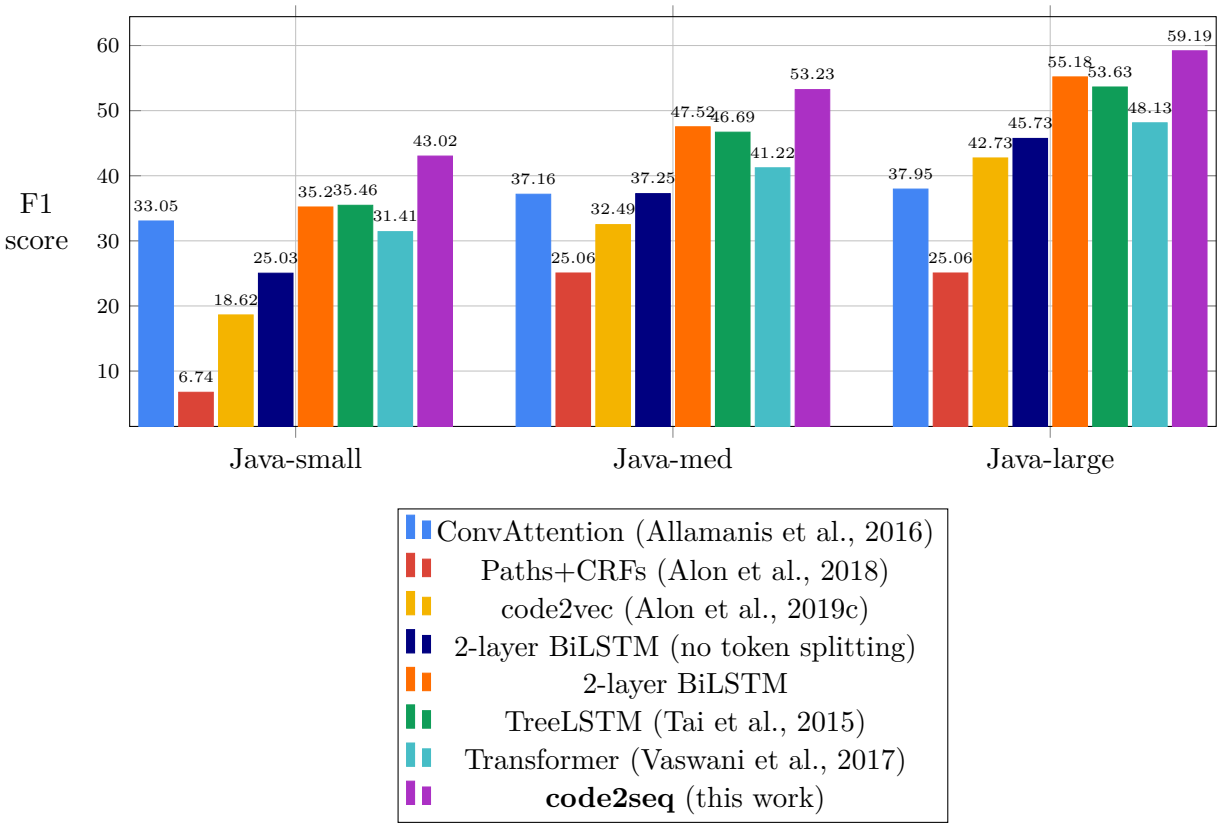


Figure 4.4: Visualization of the F1 score of our model compared to the baselines, for the code summarization task, across datasets. Our model achieves significantly higher results than the baselines.

Fernandes et al. (2019) encoded code using Graph Neural Networks (GNN), and reported lower performance than our model on Java-large without specifying the exact F1 score. They report slightly higher results than us on Java-small only by extending their GNN encoder with a subtoken-LSTM (BiLSTM+GNN→ LSTM); by extending the Transformer with GNN (SelfAtt+GNN→SelfAtt); or by extending their LSTM decoder with a pointer network (GNN→LSTM+Pointer). All these extensions can be incorporated into our model as well.

Data Efficiency ConvAttention (Allamanis et al., 2016) performed even better than the Transformer on the Java-small dataset, but could not scale and leverage the larger datasets. Paths+CRFs showed very poor results on the Java-small dataset, which is expected due to the sparse nature of their paths and the CRF model. When compared to the best among the baselines (BiLSTM with split tokens), our model achieves a relative improvement of 7.3% on Java-large, but as the dataset becomes smaller, the larger the relative difference becomes: 13% on Java-med and 22% on Java-small; when compared to the Transformer, the relative improvement is 23% on Java-large and 37% on Java-small. These results show the data efficiency of our architecture: while the data-hungry NMT baselines *require* large datasets, our model can leverage both small

and large datasets.

Sensitivity to input length We examined how the performance of each model changes as the size of the test method grows. As shown in Figure 4.5, our model is superior to all examined baselines across all code lengths. All models give their best results for short snippets of code, i.e., less than 3 lines. As the size of the input code increases, all examined models show a natural descent, and show stable results for lengths of 9 and above.

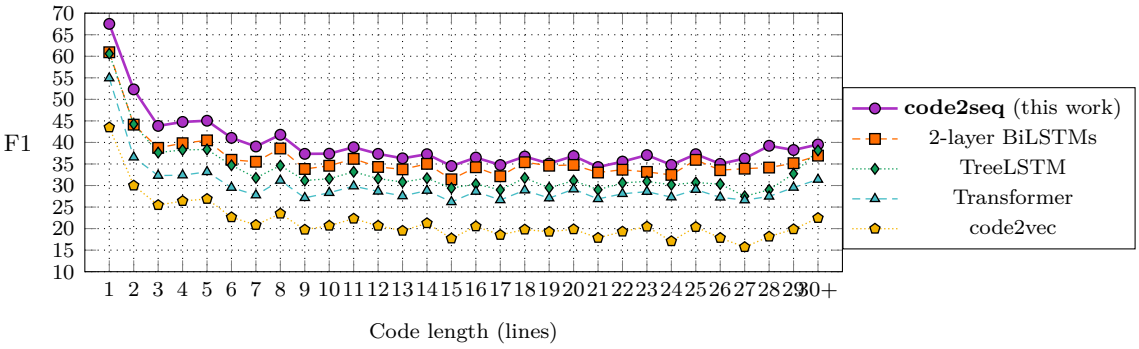


Figure 4.5: F1 score compared to the length of the input code. This experiment was performed for the code summarization task on the Java-med test set. All examples having more than 30 lines were counted as having 30 lines.

4.4.2 Code Captioning

For this task we consider predicting a full natural language sentence given a short C# code snippet. We used the dataset of CodeNN (Iyer et al., 2016), which consists of 66,015 pairs of questions and answers from StackOverflow. They used a semi-supervised classifier to filter irrelevant examples and asked human annotators to provide two additional titles for the examples in the test set, making a total of three reference titles for each code snippet. The target sequence length in this task is about 10 on average. This dataset is especially challenging as it is orders of magnitude smaller than the code summarization datasets. Additionally, StackOverflow code snippets are typically short, incomplete at times, and aim to provide an answer to a very specific question. We evaluated using BLEU score with smoothing, using the same evaluation scripts as Iyer et al. (2016).

Baselines We present results compared to CodeNN, TreeLSTMs with attention, 2-layer bidirectional LSTMs with attention, and the Transformer. As before, we provide a fair comparison by splitting tokens to subtokens, and replacing UNK during inference. We also include numbers from baselines used by Iyer et al. (2016).

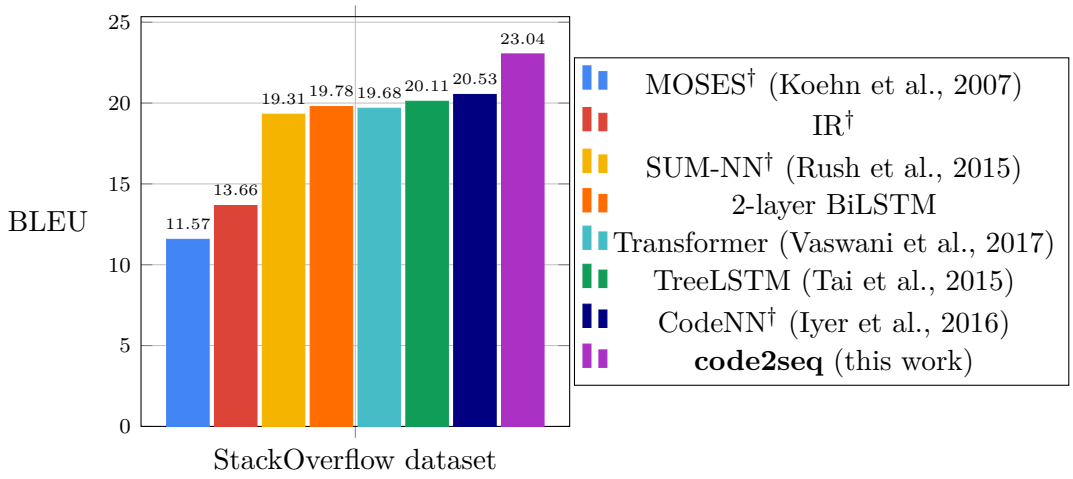


Figure 4.6: Visualization of the BLEU score of our model compared to the baselines, for the code captioning task. Our model achieves significantly higher results than the baselines.

Results Figure 4.6 summarizes the results for the code captioning task. Our model achieves a BLEU score of 23.04, which improves by 2.51 points (12.2% relative) over CodeNN, whose authors introduced this dataset, and over all the other baselines, including BiLSTMs, TreeLSTMs and the Transformer, which achieved slightly lower results than CodeNN. Examples for predictions made by our model and each of the baselines can be found in Alon et al. (2019a). These results show that when the training examples are short and contain incomplete code snippets, our model generalizes better to unseen examples than a shallow textual token-level approach, thanks to its syntactic representation of the data. Although TreeLSTMs also represent the data syntactically, the TreeLSTM baseline achieved lower scores.

4.4.3 Code Documentation

Although the task of generating code documentation is outside the focus of this work, we performed an additional comparison to Hu et al. (2018). They trained a standard seq2seq model by using the linearized AST as the source sequence and a Javadoc natural language sentence as the target sequence. While they originally report a BLEU score of 38.17, we computed their BLEU score using prediction logs provided us by the authors and obtained a BLEU score of 8.97, which we find more realistic. Training our model on the same dataset as Hu et al., matching LSTM sizes, and using the same script on our predictions yields a BLEU score of 14.53, which is a 62% relative gain over the model of Hu et al. (2018). This shows that our structural approach represents code better than linearizing the AST and learning it as a sequence.

Table 4.1: Variations on the code2seq model, performed on the validation set of Java-med.

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

4.5 Ablation Study

To better understand the importance of the different components of our model, we conducted an extensive ablation study. We varied our model in different ways and measured the change in performance. These experiments were performed for the code summarization task, on the validation set of the Java-med dataset. We examined several alternative designs:

1. *No AST nodes* – instead of encoding an AST path using an LSTM, take only the first and last terminal values to construct an input vector
2. *No decoder* – no sequential decoding; instead, predict the target sequence as a single symbol using a single softmax layer.
3. *No token splitting* – no subtoken encoding; instead, embed the full token.
4. *No tokens* – use only the AST nodes without using the values associated with the terminals.
5. *No attention* – decode the target sequence given the initial decoder state, without attention.
6. *No random* – no re-sampling of k paths in each iteration; instead, sample in advance and use the same k paths for each example throughout the training process.

Table 4.1 shows the results of these alternatives. As seen, *not encoding AST nodes* resulted in a degradation especially in the precision: a decrease of 5.16 compared to 4.30 for the recall. It is quite surprising that this ablation was still better than the baselines (Figure 4.4): for example, the Transformer can implicitly capture pairs of tokens using its self-attention mechanism. However, *not all tokens are AST leaves*. By focusing on AST leaves, we increase the focus on named tokens, and effectively ignore functional tokens like brackets, parentheses, semicolons, etc. Transformers can

(in theory) capture the same signal, but perhaps they require significantly more layers or a different optimization to actually learn to focus on those particular elements. The AST gives us this information for free without having to spend more transformer layers just to learn it. Additionally, for practical reasons we limited the length of the paths to 9 . This leads to pairs of leaves that are close in the AST, but not necessarily close in the sequence. In contrast, the Transformer's attention is effectively skewed towards sequential proximity because of the positional embeddings.

Using a single prediction with *no decoder* reduces recall by more than one-third. This shows that the method name prediction task should be addressed as a sequential prediction, despite the methods' relatively short names. Using *no token splitting* or *no tokens* at all drastically reduces the score, showing the significance of encoding both subtokens and syntactic paths. Despite the poor results of *no tokens*, it is still surprising that the model can achieve around half the score of the full model, as using no tokens is equivalent to reasoning about code which has no identifier names, types, APIs, and constant values, which can be very difficult even for a human. The *no attention* experiment shows the contribution of attention in our model, which is very close in its relative value to the contribution of attention in seq2seq models (Luong et al., 2015; Bahdanau et al., 2014). The *no random* experiment shows the positive contribution of sampling k different paths afresh on every training iteration, instead of using the same sample of paths from each example during the entire training. This approach provides data-level regularization that further improves an already powerful model.

Chapter 5

Structural Language Models of Code

5.1 Introduction

Code completion is the problem of generating code given its surrounding code as context. In its most general form, this problem is extremely challenging as it requires reasoning over an unbounded number of syntactic structures and user-defined symbols. Previous approaches have avoided this issue by limiting the generation problem: program synthesis approaches are often tailored to domain-specific languages (Gulwani, 2011; Polozov and Gulwani, 2015; Devlin et al., 2017; Ellis et al., 2019), while other recent approaches generate code in general languages like Java and C#, but severely restrict the syntax, vocabulary, domain, or nature of the generated programs (Murali et al., 2018; Brockschmidt et al., 2019; Young et al., 2019).

We introduce the task of *any-code completion* – generating code in a general-purpose programming language without any restriction on its vocabulary or structure. Specifically, we focus on generating code in context: given a program $\mathcal{P}$ and some part of the program p, the task is to predict p from the rest of the program $\mathcal{P}^{-}{=}\mathcal{P}\backslash p$. Any-code completion thus generalizes the restricted completion task of Brockschmidt et al. (2019), in which the target code contained only primitive types (e.g., `int` and `string`) and excluded user-defined functions. Figure 5.1 shows two any-code completion examples.

In related tasks such as semantic parsing (Dong and Lapata, 2018; Yu et al., 2018; Iyer et al., 2019), natural-language-to-code (Allamanis et al., 2015b; Iyer et al., 2018), and edit-to-code (Yin et al., 2019; Zhao et al., 2019), models must use separate encoders and decoders because of the different modalities of the input (e.g. natural language text) and the output (code). In contrast, we leverage the fact that our input and output are of the *same modality* (code), and pursue better generalization by modeling them *jointly*.

In Alon et al. (2020), we present a new approach that explicitly models the source

```
public static Path[] stat2Paths(
    FileStatus[] stats) {
  if (stats == null) return null;
  Path[] ret = new Path[stats.length];
  for (int i = 0; i < stats.length; ++i){
    ret[i] = [          ];
  }
  return ret;
}
```

True ref:		stats[i].getPath()
	(25.2%)	stats[i].getPath()
SLM	(3.3%)	Path(stats[i])
top-5:	(2.5%)	new Path(stats[i], charset)
(Java)	(1.7%)	stat(stats[i], ret)
	(0.8%)	new Path(stats[i])

(a)

```
public static string Camelize(
    this string input)
{
    var word = input.Pascalize();
    return word.Length > 0 ?
        [          ].ToLower()
          + word.Substring(1)
        : word;
}
```

True ref:		word.Substring(0, 1)
	(14.1%)	word.Substring(0, 1)
SLM	(8.2%)	word.trim()
top-5:	(5.8%)	word.Substring(1)
(C#)	(2.4%)	input.Substring(0, 1)
	(1.9%)	wordValue.Substring(0, 1)

(b)

Figure 5.1: Examples from the Java (left) and C# (right) test sets. The highlighted expression in each example is the target p, which our models correctly generated from the rest of the snippet. Additional and larger examples can be found in the supplementary material.

and the target code as the same tree – *structural language modeling* (SLM). SLM estimates the probability of the program's abstract syntax tree (AST) by decomposing it into a product of conditional probabilities over its *nodes*. We present a neural model that computes these conditional probabilities by considering all AST paths leading to a target node, generalizing over traditional language models that consider sequences of words. While prior work uses AST paths to *read* programs (Alon et al., 2019c), we *generate* code by predicting the next node along the set of paths, generating the target AST node-by-node.

We evaluate SLMs on Java any-code completion, achieving a new state of the art: exact-match accuracy@1 of 18.04% and accuracy@5 of 24.83% (previous SOTA: 16.93% and 23.17%). SLMs also outperform existing models in the restricted completion task of Brockschmidt et al. (2019) in C# by a wide margin, 37.61% accuracy@1 compared to 26.42%. Our ablation study reveals the importance of *joint modeling* of the source and target code, rather than separating encoders from decoders. Finally, we discuss the theoretical advantages of SLMs, and show how they generalize many previous structural approaches for code generation. An interactive demo of our model is presented at `http://AnyCodeGen.org`.

5.2 Code Generation as Structural Language Modeling

We model the task of any-code completion by computing the probability of a program $Pr(\mathcal{P})$, similar to how a language model computes the probability of a natural language sentence. While language models typically assume a *sequence* as their input, our input is an abstract syntax *tree* $\mathcal{A}_\mathcal{P}$. We thus introduce a *structural* language

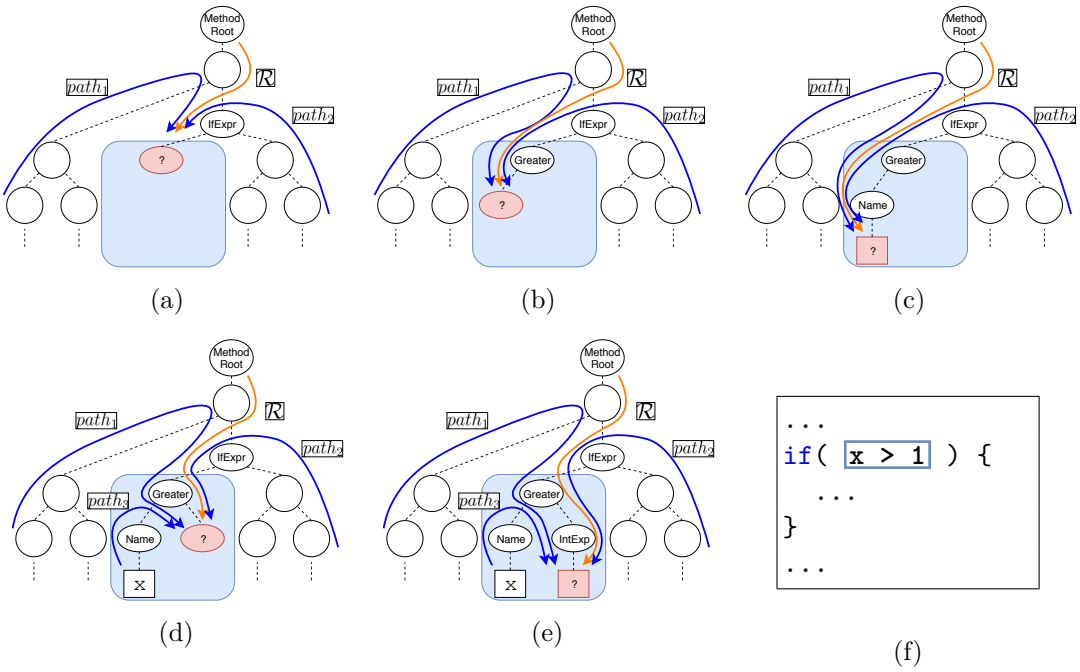


Figure 5.2: The subtree representing `x > 1` is generated given its surrounding tree. At each step, the model generates the next node (denoted by ?) of $path_1$, $path_2$ and $path_3$ using the root path $\mathcal{R}$. Dashed lines denote the AST structure; solid lines denote AST paths. Most AST paths are omitted from the figure, for clarity.

modeling approach (SLM).

The intuition behind this idea is that a language model could *generalize better* by modeling the *tree* rather than the sequential form of the program. Further, learning from the AST allows a model to save learning capacity, instead of having to re-learn known syntactic patterns from the text.

We first show a chain-rule decomposition of the tree's probability $Pr(\mathcal{A}_\mathcal{P})$ into a product of conditional *node* probabilities, and then describe our path-based model for computing the individual conditional probabilities. We explain how to construct a tree from local node predictions, and finally discuss how our approach differs from previous work on production-based tree generation.

Representing Code as a Tree A program $\mathcal{P}$ is a sequence of tokens that can be unambiguously mapped to an abstract syntax tree (AST) $\mathcal{A}_\mathcal{P}$, where every node represents an element in the language (e.g. conditions, loops, variable declarations) from a set $\mathcal{T}$. Each AST leaf (terminal) has an associated user-defined value $v \in \mathcal{V}$. Nonterminal nodes can have a varying number of children nodes.

Decomposing the Probability of a Tree Given a tree $\mathcal{A}_\mathcal{P}$, we first traverse the tree, depth-first,[1] to induce an ordering over its nodes $a_0, \ldots, a_{|\mathcal{A}_\mathcal{P}|} \in \mathcal{A}_\mathcal{P}$. We de-

[1]Depth-first ordering is a common practice in tree generation (Maddison and Tarlow, 2014; Raychev et al., 2016b), but in principle our framework also allows for other orderings.

compose the probability of a tree $Pr(\mathcal{A}_\mathcal{P})$ using the chain rule, akin to the standard approach in language modeling:

$$Pr(\mathcal{A}_\mathcal{P}) = \prod_t Pr(a_t | a_{<t}) \tag{5.1}$$

where $a_{<t}$ are all the nodes that were traversed before a_t.

In any-code completion, part of the tree ($\mathcal{A}_{\mathcal{P}^-}$) is already observed. Therefore, we order the nodes of $\mathcal{A}_{\mathcal{P}^-}$ to be before the nodes of the target p, and compute only the conditional probabilities over the nodes in p, essentially conditioning on the observed tree $\mathcal{A}_{\mathcal{P}^-}$.

Representing Partial Trees via Paths How can we represent the partial tree composed of $a_{<t}$ when computing $Pr(a_t | a_{<t})$? In standard language modeling, the structure is linear, and $a_{<t}$ is a sequence. One way to represent a partial tree is to linearize it according to the traversal order (Xiao et al., 2016); however, this creates artificially long distances between the current node a_t and ancestor nodes (e.g., the root a_0). Another option is to use only the path from the root node to a_t (Rabinovich et al., 2017), but this ignores a lot of contextual information (e.g., sibling nodes).

We follow Alon et al. (2018) and use *the set of paths* from every leaf to a_t together with the path from the root to a_t. Intuitively, each path captures the effect of a different, possibly distant, program element on a_t, along with the syntactic relationship between them. For example, in Figure 5.1 (left) the three paths originating from `Path[] ret` inform the model about the existence of `ret` which is an array of type `Path`. Thus, when completing `ret[i] = ...` – the completion should be a `Path` object. Other paths inform the model that the target is inside a `For` loop, iterated `stats.length` times. Considering the information flowing from all paths, our model correctly generates `stats[i].getPath()`.

We denote the (candidate) node at time t as a_t, its (given) parent, which is currently expanded, by $\pi(a_t)$, and the set of all paths as $\mathcal{S}_t$:

$$\mathcal{S}_t = \{\ell \rightsquigarrow \pi(a_t) \,|\, \ell \in \text{leaves}(a_{<t})\}$$

where $\ell \rightsquigarrow \pi(a_t)$ is the (only) path in the tree between a leaf ℓ and the current node to expand $\pi(a_t)$. We denote the path from the root of the program as $\mathcal{R}_t = a_0 \rightsquigarrow \pi(a_t)$, which represents the current, relative position of $\pi(a_t)$ in the program (marked as $\mathcal{R}$ in Figure 5.2). Whereas prior work used *whole* paths (between two leaf nodes) to encode ASTs (Alon et al., 2019a,c), our model observes *partial* paths (between a leaf and any other node) and learns to extend them by predicting their next node.

Figure 5.2 illustrates the traversal order of a subtree that represents the expression `x > 1` and some of the paths used to compute the probability at each step. At each step, the probability of the next node is computed given the paths $\mathcal{S}_t$ from the root

and every given leaf up to the current node to expand. Figure 5.2d shows how after the terminal node with the value `x` is given, $path_3$ originating from this leaf is also used to compute the probability of the next nodes.

Our path-based approach generalizes previous approaches such as "parent feeding" and "previous action" encoding (Yin and Neubig, 2017), context nodes (Bielik et al., 2016), and some of the graph-edges of Brockschmidt et al. (2019). See Section 5.8 for further discussion.

Generating Trees In sequence generation, the length of the target sequence is controlled by generating an `EOS` token to stop. When generating trees, we require a more sophisticated mechanism to control arity and depth. We augment $\mathcal{A}_\mathcal{P}$ in two ways to allow node-by-node generation.

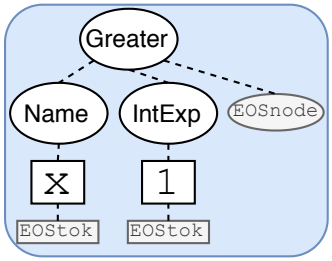


Figure 5.3: Augmenting the AST with $\texttt{EOS}_{node}$ and $\texttt{EOS}_{tok}$ nodes.

First, we add a special $\texttt{EOS}_{node}$ node to every nonterminal to control for *arity*. Generating this node indicates that the parent node has no more children nodes. Second, we end each subtoken sequence with a special $\texttt{EOS}_{tok}$ node to control for *depth* during generation; we decompose each terminal node n_v into a sequence of terminal nodes T_v by splitting up the node's value v into *subtokens* based on camel notation. For example, if $v = \texttt{toLowerCase}$, then $T_v = \texttt{to} \rightarrow \texttt{lower} \rightarrow \texttt{case} \rightarrow \texttt{EOS}_{tok}$. Figure 5.3 shows an example of both $\texttt{EOS}_{node}$ and $\texttt{EOS}_{tok}$ in action.

Node Trees vs. Production Trees While we predict a single *node* at each step, previous work (Iyer et al., 2018, 2019) predicts a grammar production rule. This representation decomposes the code in a way that often forces the model to predict with partial information. For instance, consider generating the expression `str.Substring(3)`. The model of Brockschmidt et al. (2019) would first predict the rule `Expr→Expr.Substring(Expr)`, and only then expand `Expr→str` and `Expr→3`. That is, the model needs to predict the method name (`Substring`) *before* the invoking object (`str`). Further, the `Substring` method can get either one *or* two arguments, forcing the model to choose whether to use the one- or two-argument rule in advance. Node generation, however, allows us to predict the presence of a function call and only then to predict its object and method name, rather than predicting these a priori.

5.3 Model Architecture

In the previous section, we described how we can generate code given the probabilities $Pr\left(a_t | a_{<t}\right)$, where $a_{<t}$ is represented by the set of partial AST paths $\mathcal{S}_t$. Here, we present a neural model that estimates $Pr\left(a_t | \mathcal{S}_t\right)$. We first encode each path in $\mathcal{S}_t$ as a

vector (Section 5.3.1); then, we contextualize and aggregate the entire set. Finally, we predict the target node a_t by combining a subtoken vocabulary with a syntactic copy mechanism (Section 5.3.3).

5.3.1 Encoding AST Paths

Given a partial AST path, i.e., a sequence of nodes $n_1, \ldots, n_k$, our goal is to create a vector representation.

We first represent each node n_i using embeddings. A subtoken node is represented by the index of its subtoken w in the embedding matrix E^{subtoken}; AST nodes are represented as a pair $n_i = (\tau, \kappa)$ where τ is the node type, e.g. `IfStatement`, and κ is the node index among its sibling nodes. We represent node types using a learned embedding matrix E^{type} and the child indices using a learned matrix E^{index}. The node's vector representation is the concatenation of the type and index vectors.

$$e\left(n_i\right) = \begin{cases} E_w^{\text{subtoken}} & n_i \text{ is the subtoken } w \\ \left[E_\tau^{\text{type}}; E_\kappa^{\text{index}}\right] & n_i \text{ is the AST node } (\tau, \kappa) \end{cases}$$

We encode the entire path using a uni-directional LSTM stack, and take the final states:[2]

$$\overrightarrow{f}\left(n_1, \ldots, n_k\right) = \text{LSTM}\left(e\left(n_1\right), \ldots, e\left(n_k\right)\right)$$

Given a set of partial paths $\mathcal{S}$ (omitting the iterator t for simplicity), we denote their encodings as $H = \{\overrightarrow{f}\left(n_1, \ldots, n_k\right) \mid \left(n_1, \ldots, n_k\right) \in \mathcal{S}\}$.

Efficient Computation When modeling a subtree, there are large overlaps between paths from different time steps. In particular, paths that originate from the same leaf share the same *prefix*. We therefore apply the LSTM on the prefix *once* and cache the intermediate state across suffixes, speeding up both training and inference significantly. An example is shown in the supplementary material (Fig. 2).

5.3.2 Aggregating Multiple Paths

Given the set of paths $\mathcal{S}$ leading up to the parent $\pi(a)$ of the target node a, our goal is to represent $\mathcal{S}$ in the context of predicting a. To do so, we introduce the aggregation function $g\left(H, r, i\right)$. As its input, g takes the set of encoded paths H, the encoded root path r, and the child index i of the currently predicted child node a relative to its parent.

We first contextualize the path encodings H using a transformer encoder (Vaswani et al., 2017).[3] In parallel, we apply a non-linear transformation to the encoding of the

[2]Replacing the LSTMs with transformers yielded similar results in preliminary experiments.

[3]Since H is a set, we do not use positional embeddings.

root path $r = \overrightarrow{f}(\mathcal{R})$, in order to inform it that we wish to predict the i-th child of $\pi(a)$:

$$Z = \text{Transformer}(H) \qquad \widetilde{r} = W_a \cdot \text{ReLU}(C_i \cdot r)$$

In this formulation, the parameter matrix C_i is used when the child index is i, while the parameter matrix W_a is used for every instance.

We then compute attention over the set of contextualized path encodings Z using the index-informed root-path encoding $\widetilde{r}$ as the query; we pass the weighted average $\widetilde{z}$ and the root-path encoding $\widetilde{r}$ through another fully-connected layer; we denote the resulting vector representation as $\widetilde{h}$:

$$\boldsymbol{\alpha} = \text{softmax}(Z \cdot \widetilde{r}) \qquad \widetilde{z} = \sum_j \alpha_j \cdot Z_j \tag{5.2}$$

$$\widetilde{h} = g(H, r, i) = \text{ReLU}(W_g[\widetilde{z}; \widetilde{r}])$$

where semicolons (;) denote vector concatenation.

5.3.3 Predicting with a Syntactic Copy Mechanism

We can now predict a from the representation $\widetilde{h}$. If the target node's parent $\pi(a)$ is a nonterminal AST node, then a must be an AST node; otherwise, a is a subtoken.

Predicting AST Nodes If a is an AST node, we predict a using a softmax over the node type embeddings E^{type}:

$$Pr(a|\mathcal{S}) = \text{softmax}\left(E^{\text{type}} \cdot \widetilde{h}\right) \qquad (\pi(a) \text{ is a nonterminal})$$

Predicting Subtokens Programs repeatedly refer to previously declared symbols, resulting in highly repetitive usage of identifiers. We therefore use a copy mechanism (Gu et al., 2016) to allow our model to predict either entire tokens or individual subtokens that exist in the context. As we show in Section 5.6, copying greatly improves our model's performance. For brevity, we describe how entire tokens are copied, and elaborate on the copy of *sub*tokens in the supplementary material. We score each leaf ℓ using a bilinear function (W_c) between its path's encoding H_ℓ and $\widetilde{h}$. At the same time, we score the token w, which is the token associated with ℓ, from a limited vocabulary using the inner product between its representation in the subtoken embedding matrix E^{subtoken} and $\widetilde{h}$.

$$s_{\text{copy}}(\ell) = H_\ell \cdot W_c \cdot \widetilde{h} \qquad s_{\text{gen}}(w) = E_w^{\text{subtoken}} \cdot \widetilde{h}$$

The scores s_{copy} and s_{gen} are then summed over all occurrences that correspond to the same symbol and subsequently normalized via softmax.

A key difference from most previous work (Ling et al., 2016; Yin and Neubig, 2017) is that our copy mechanism uses the *syntactic* relation to the source (the path H_ℓ), rather than the sequential relation or the graph-node representation (Yin et al., 2019).

5.4 Experimental Setup

5.4.1 Benchmarks

Any-Code Completion: Java We take the Java-small dataset of Alon et al. (2019a), which is a re-split of the dataset of Allamanis et al. (2016). It contains 11 GitHub projects, broken down into a single method per example, and split to train/dev/test by project to reduce code overlap. This dataset was found to contain the least code duplication by Allamanis (2018b). We create any-code completion examples by selecting every expression larger than a single AST node as the target, using the remainder of the method as the context. We remove methods containing the word "test" in their body or file name, and omit 10% of the examples by filtering out methods longer than 20 lines to avoid configurations, initializations, and auto-generated code. To make the task even harder, we remove examples where the target appears as-is in the context. Ultimately, this dataset contains 1.3M/10k/20k train/dev/test examples.

Restricted Completion: C# To provide a fair comparison to Brockschmidt et al. (2019), we create an additional benchmark where the missing code is more limited. We use the code of Brockschmidt et al. (2019) which filters out examples where the targets contain non-primitive types or user-defined functions. We extract the exact same types of limited expressions. Since the dataset of Brockschmidt et al. (2019) is not publicly available, we consulted with Brockschmidt et al. directly and extracted examples from the raw dataset of Allamanis et al. (2018) using their "unseen projects test" set. This dataset contains 30 GitHub projects broken down to one method per example. This dataset contains 16k/8k/3k train/dev/test examples.

Our datasets are available at: `http://github.com/tech-srl/slm-code-generation/`. Detailed statistics are provided in the supplementary material.

Metrics Following Brockschmidt et al. (2019), we report exact match accuracy at 1 and 5. We also introduce a new *tree@k* metric which counts a prediction as correct if the entire tree structures, ignoring leaf values, are identical. For example, `x > 1` and `y > 2` would *not* count as identical in *exact match*, but *would* count as "tree-match identical" because both express that an identifier is greater than an integer (`NAME > INT`). The *tree@k* metric is interesting because it allows us to tease apart the model's syntactic errors from incorrect subtoken predictions.

Model	acc@1	acc@5	tree@1	tree@5
code2seq (Alon et al., 2019a)	10.68	15.56	30.46	43.94
Iyer et al. (2018)	5.94	9.19	25.54	36.75
seq2prod (Yin and Neubig, 2017)	8.05	11.82	30.77	41.73
$\text{Transformer}_{\text{small}}$ (Vaswani et al., 2017)+copy	14.23	21.35	31.83	47.40
$\text{Transformer}_{\text{base}}$ (Vaswani et al., 2017)+copy	16.65	24.05	34.68	50.52
BiLSTM→LSTM (Luong et al., 2015)+copy	16.93	23.17	34.29	49.72
seq2tree (Aharoni and Goldberg, 2017)+copy	16.81	23.04	38.14	52.36
SLM (this work)	**18.04**	**24.83**	**39.10**	**55.32**

Table 5.1: Results on any-code completion in Java.

5.4.2 Baselines

We compare our model to a variety of original implementations and adaptations of existing models. We put significant effort to perform a fair comparison, including adding a copy mechanism to the NMT baselines and *sub*tokenization as in our model. We adapt strong baselines from the literature to our task, even if they were designed to different tasks such as NL→code and code→NL. We re-train all the following baselines on the same datasets as our models.

NMT We use standard autoregressive sequence-to-sequence NMT baselines, in which we subtokenize the given code snippet, replace the target in the source with a special `PRED` symbol, and train the network to predict the target as a sequence of subtokens. *Transformer$_{base}$+copy* (Vaswani et al., 2017) uses the implementation of OpenNMT (Klein et al., 2017) with a copy mechanism (Gu et al., 2016). *Transformer$_{small}$+copy* uses d_{model}=256, d_{ff}=1024, and 4 self attention heads per layer. *BiLSTM→LSTM+copy* is a 2-layer bidirectional LSTM encoder-decoder with d=512 and attention. *seq2tree+copy* follows Aharoni and Goldberg (2017) and learns to generate the linearized, subtokenized target AST.

Java-specific Baselines We use the original implementation of Iyer et al. (2018), and also their *seq2prod* baseline which is a re-implementation of Yin and Neubig (2017); these are designed for NL→code tasks, in which we feed the code context as the NL input. The model of Iyer et al. (2018) is designed to get additional input of the available variables *and their types*, for which we do not feed types. While these models could also be applied to other languages, their implementation only supports Java.

C#-specific Baselines We compare our model to the graph-based *GNN→$\mathcal{NAG}$* model using the implementation of Brockschmidt et al. (2019). Bielik et al. (2016) kindly trained and tested their non-neural PHOG model on our C# dataset. We note that PHOG does not have an explicit copy mechanism, and considers only context

to the left of the target code, while we consider also context to the right. Extending PHOG could potentially improve its results.

In both Java and C#, we compare to *code2seq* (Alon et al., 2019a), which is a strong code→NL model. We train it to generate the target code as a *sequence* of subtokens.

5.4.3 Implementation and Hyperparameter Settings

Architecture We use embeddings of size 512, 2 layers of LSTMs with 256 units, and 4 transformer layers with 8 attention heads. We kept a small subtoken vocabulary of size 1000 to encourage the model to learn to copy; larger vocabularies did not show an improvement. These resulted in a very lightweight model of only 15M parameters, which is close to $\mathit{Transformer}_{\mathit{small}}$ (11.8M parameters). In comparison, $\mathit{Transformer}_{\mathit{base}}$ had more than 45M parameters (3× more parameters than our model).

Training We train the model end-to-end on a single V100 GPU, using cross entropy and the Adam optimizer (Kingma and Ba, 2014), an initial learning rate of 10^{-4} multiplied by 0.95 every $20k$ steps. We bucket examples based on the number of predictions in the target subtree (nodes + subtokens + `EOS`), and vary the batch size such that each batch contains about 512 targets. We train the model to prefer copying entire tokens rather than copying subtokens, if possible, by optimizing for the entire token as the true label. We apply dropout of 0.25 in the Transformer layers, and a recurrent dropout of 0.5 in the LSTMs.

Inference We perform beam search with width of 5 and optimize for accuracy@1.

Model	acc@1	acc@5	tree@1	tree@5
GNN→NAG	15.19	27.05	26.48	40.09
code2seq	6.20	10.05	21.97	30.89
seq2seq+copy	26.42	37.94	34.10	49.23
seq2tree+copy	22.29	35.86	31.85	48.53
PHOG	7.40	12.00	–	–
SLM (this work)	**37.61**	**45.51**	**51.10**	**59.82**

Table 5.2: Results on restricted completion in C#.

5.5 Results

Any-Code Completion: Java Table 5.1 shows that our SLM achieves over 1.1% and 0.78% better *acc@1* and *acc@5* (respectively) over the two strongest baselines. The improvement over $\mathit{Transformer}_{\mathit{small}}$, which is closer to our model in the number of parameters, is even higher: over 3.8% and 3.4% in *acc@1* and *acc@5*.

Ablation	acc@1	acc@5	tree@1	tree@5
Paths→Seq	12.95	18.52	33.44	43.43
Seq→Path	12.12	17.12	28.68	43.99
Paths→Paths	17.63	24.62	37.78	53.98
No Root Att	14.43	18.48	28.20	35.65
No Copy	10.72	15.70	30.61	44.35
SLM (original)	18.04	24.83	39.10	55.32

Table 5.3: Ablations on any-code completion in Java.

The NMT baselines performed better than code-specific baselines. We hypothesize that the reason is that the NMT baselines are more generic, while the code-specific baselines are designed for different tasks: *seq2prod* is designed for tasks which involve generating code *given natural language input*; Iyer et al. (2018) additionally expects all member methods, fields, and their types as input; *code2seq* is designed to generate sequences rather than code, and does not have a copy mechanism. An approximation of *code2seq* with a copy mechanism is presented in Section 5.6.

Interestingly, the syntactically-informed *seq2tree* baseline achieved the highest *tree@k* among the baselines, while our model achieved higher *acc@k* and *tree@k*. This shows that leveraging the syntax can benefit NMT models as well.

Restricted Completion: C# Table 5.2 shows the results for the restricted completion task in C#, where *seq2seq+copy* is the *BiLSTM→LSTM+copy* model which performed the best among the Java baselines. We first observe that the *seq2seq+copy* and the *seq2tree+copy* baselines outperform the *GNN→*$\mathcal{NAG}$ of Brockschmidt et al. (2019), who introduced this task. Although Brockschmidt et al. (2019) did compare to a seq2seq baseline, their *GNN→*$\mathcal{NAG}$ model could copy symbols from the context, but their baseline did not. To conduct a fair comparison with our SLM model, we equipped the seq2seq and seq2tree baselines with a copy mechanism. Even though the *seq2seq+copy* and the *seq2tree+copy* baselines perform substantially better than the state of the art in this setting, our SLM model is able to go beyond, achieving significant gains over all models.

The superiority of our model over *GNN→*$\mathcal{NAG}$ may also be related to the GNN bottleneck (Alon and Yahav, 2021), which hinders GNNs from propagating long-range messages. In contrast, propagating long-range messages using paths is natural for our model.

5.6 Ablation Study

To understand the importance of the various components and design decisions in our model, we conducted an extensive ablation study.

Paths→Seq follows *code2seq* (Alon et al., 2019a) and separates the model to an encoder and a decoder, where the decoder generates the target code as a sequence of subtokens. The main difference from *code2seq* is that *Paths→Seq* includes a copy mechanism, as in our model.

Seq→Path follows Rabinovich et al. (2017) and separates our model to an encoder and a decoder (including a copy mechanism), where the encoder encodes the context as a sequence of subtokens using a BiLSTM, and the decoder generates the missing subtree using the root path and the index of the generated child.

Paths→Paths is similar to our SLM model except that it uses separate encoder and decoder. These encoder and decoder have untied weights, unlike our SLM model which models the source and the target jointly.

No Root Attention uses max pooling instead of attention in aggregating multiple paths (see Section 5.3.2). The index-informed path from the root to the target's parent ($\mathcal{R}$ in Figure 5.2) is concatenated with the result, instead of being used as attention query.

No Copy replaces copy mechanism with a much larger vocabulary (25k subtokens instead of 1k).

Results Table 5.3 shows the results of these alternatives. As our SLM model performs better than *Paths→Paths*, this ablation shows the importance of joint modeling of the context and the target subtree by parameter tying.

Each of *Paths→Paths* and the seq2seq baselines (Table 5.1) performs better than *Paths→Seq* and *Seq→Path*; this shows the importance of *using the same type of encoder and decoder* for any-code completion, rather than combining "an optimal encoder" with "an optimal decoder". While this distinction between encoder and decoder types might be necessary for semantic parsing (Rabinovich et al., 2017; Dong and Lapata, 2018), NL→code (Yin and Neubig, 2017) and code→NL (Alon et al., 2019a; Fernandes et al., 2019) tasks because of the different modalities of the input and the output, this discrepancy may hurt generalization when the output is essentially a missing part of the input's AST.

Paths→Paths performs better than the seq2seq baselines (Table 5.1), showing the advantage of using paths over textual sequences, even without parameter tying.

No Root Attention degrades *acc@1* and *acc@5* by 3.6% to 6.3%. This shows that dynamically attending to the context paths given the current root path is crucial.

Not using a copying mechanism results in a degradation of 7.3% to 9.1%. Programs use symbols and identifiers repetitively, thus the ability to copy symbols from the context is crucial for this task. For this reason, we included a copying mechanism in all NMT baselines in Section 5.4.

5.7 Qualitative Analysis

```
private static void log(String value) {
  if (value != null
      && [          ])
    value = value.substring(0, 55)+"...";
  LOG.info(value);
}
```

```
public int compareTo(LongWritable o) {
    long thisValue = this.value;
    long thatValue = o.value;
    return (thisValue < thatValue ? -1 :
        ([                    ]));
}
```

True ref:	`value.length() > 55`	
SLM top-5:	`value.length() > 0`	🌳
	`value.length() > 55`	**✓**
	`value.startsWith("...")`	
	`!value.startsWith("...")`	
	`value.charAt(0) == '.'`	

(a)

`thisValue == thatValue ? 0 : 1`	
`thisValue == thisValue ? 0 : 1`	🌳
`thisValue == thatValue ? 0 : 1`	**✓**
`thisValue == value ? 0 : 1`	🌳
`thisValue > thatValue ? 0 : 1`	
`(thisValue == thatValue) ? 0 : 1`	↔

(b)

Figure 5.4: Examples for cases where the top candidate is a "tree-match" (marked with 🌳), but only the second candidate is an "exact match" (marked with ✓ in bold). Predictions that are logically equivalent to the ground truth are marked with ↔.

Our main results (Table 5.1 and Table 5.2) reveal a gap between *acc@k* and *tree@k*: when ignoring identifier values and comparing only the tree structure, accuracy is significantly higher across all models. While our SLM model performs better than all baselines in *acc@k*, our model also shows greater potential for improvement in its *tree@k* results, which are much higher than the baselines'. We thus focus on studying the cases where the tree was predicted correctly, but the model failed to generate the code exactly including names.

Figure 5.4a shows an example of this case: the ground truth has a structure of the form: `NAME.NAME() > INT`. Our model predicts `value.length() > 0` (a tree-match) as its first candidate and `value.length() > 55` (the ground truth) as its second. Null-checking a string is often followed by checking that it is also not empty, making the first candidate a reasonable prediction as well.

Figure 5.4b shows another example: in this case, the ground truth `thisValue == thatValue ? 0 : 1` was predicted correctly only as the second candidate. Nevertheless, the top-3 candidates are tree-matches since all of them are of the form: `NAME == NAME ? INT : INT`. Interestingly, the fifth candidate `(thisValue == thatValue) ? 0 : 1` is logically-equivalent to the ground truth.

In both examples, our model's top candidate differs from the ground truth by *a single identifier or literal*: in Figure 5.4a the model predicted `0` instead of `55`; in Figure 5.4b the model predicted `thisValue` instead of `thatValue`. Such single *sub*token errors are responsible for 30% of the cases where the model's top prediction is a tree-match but not an exact match. Single *token* (whole identifier or literal) mismatches are responsible for 74% of these cases. Thus, improving our model's ability to predict the right names has the potential to enhance our gains furthermore. Detailed results of allowing such mistakes in our model and in the baselines can be found in the supplementary

```
public float getProgress() {
    this.readLock.lock();
    try {
        if (this.currentAttempt != null) {
            return                                  ;
        }
        return 0;
    } finally {
      this.readLock.unlock();
    }
}
```

True ref:		`this.currentAttempt.getProgress()`			
SLM top-5:	(31.3%)	`this.currentAttempt.getCount()`		⚙	
	(30.6%)	`-1`			⚙
	(1.5%)	`this.currentAttempt.get()`		⚙	
	(1.2%)	`this.currentAttempt.getTime()`		⚙	
	(0.9%)	**`this.currentAttempt.getProgress()`**	**✓**	⚙	⚙

Figure 5.5: An example from our test set in which a compiler-guided generation could filter out non-compiling candidates, and thus rank the ground truth *second* instead of *fifth*. Four out of the five candidates are "tree-match" (marked with ⚙), the fifth candidate is an "exact match" (marked with ✓ in bold), and only the second and the fifth candidate compile (marked with ⚙).

material.

Additional possible post-filtering could filter out candidates that do not compile. In Figure 5.5, the first, third and fourth candidates do not compile, because the `this.currentAttempt` object does not have `getCount`, `get`, nor `getTime` methods. If the model's predictions would have been considered in the context of the entire project including its dependencies, these candidates could have been filtered out, and the (correct) fifth candidate would be ranked *second*. We leave compiler-guided code generation to future work.

Additional examples can be found in the supplementary material and in our interactive demo at `http://AnyCodeGen.org`.

5.8 Related Work

Generalizing Previous Approaches Our approach frames code generation as predicting the next node in all partial AST paths. This simple framing generalizes most previous work, without hand-crafted edges and special actions:

- Models that use information about ancestor nodes only (Rabinovich et al., 2017), as well as the "Parent Feeding" of Yin and Neubig (2017), are generalized by our model, since all paths that go into a node a_t pass through its parent, and the path from the root is the attention query.
- The "previous action encoding" of Yin and Neubig (2017) is also a special case of our approach, because $\mathcal{S}_t$ contains the paths starting from the *previously expanded*

leaves of $\mathcal{A}_p$ into the currently expanded node $\pi(a_t)$, such as $path_3$ in Figure 5.2e.

- The "context node" of PHOG (Bielik et al., 2016) is just one of the previously-traversed leaf nodes in $a_{<t}$. Thus, not only that our model conditions on this context node as well, our model also takes into account the *syntactic relation*, i.e., the path, between the context and $\pi(a_t)$. Moreover, while PHOG conditions on a single leaf, SLMs condition on *every* leaf in $a_{<t}$.
- Finally, Brockschmidt et al. (2019) define special graph edges (e.g., "NextSib" and "Child") to capture relations on the AST. Allamanis et al. (2018) further defines data-flow and control-flow graph edges such as "ComputedFrom" and "GuardedByNegation". Most of these relations can be expressed as partial AST paths without manually designing them.

Program Generation Learning to generate programs is one of the oldest problems in machine learning (Waldinger and Lee, 1969) and has been considered by some as the "holy grail of computer science" (Pnueli and Rosner, 1989; Gulwani et al., 2017). Typically, the task is to generate a program given some form of input or context, such as complete formal specifications (Green, 1981; Si et al., 2019) or input-output examples (Gulwani, 2011; Devlin et al., 2017; Parisotto et al., 2017; Balog et al., 2017; Gaunt et al., 2017).

While these approaches work well in some cases, they are often bounded to DSLs that prevent them from being applied to realistic, general-purpose code.

Bielik et al. (2016) learn a dynamic DSL expression that points to a *single* context that guides the generation of a JavaScript program. Maddison and Tarlow (2014) and Amodio et al. (2017) generate general-purpose unconditional code, and do not deal with the challenge of fitting the code to a given context.

Brockschmidt et al. (2019) addressed a similar code completion task as ours using a graph encoder and a neural attribute grammar decoder. However, they limit their model to generate only primitive types or arrays of these; use a closed vocabulary; and omit user-defined functions. In this work, we lift these constraints and allow any, general-purpose, generation of code, of all types and containing any names. As we show in Section 5.5, our model performs significantly better.

Murali et al. (2018) generate code given a set of APIs in a "Java-like" language; they state that their approach is thus intrinsically limited to generate only API-heavy programs. Yin et al. (2019) generate general-purpose code by applying a given edit to a given code snippet. Brody et al. (2020) predict code edits directly given other edits that occurred in the context. Yin and Neubig (2017) and Rabinovich et al. (2017) used a top-down syntactic approach for generating general-purpose code given a natural language description. Models that address APIs→code, edit→code, or NL→code tasks must model the input separately and differently from the output code. As we show in Section 5.6, modeling the source and the target differently perform poorly in our task, in which the input is code as well.

Chen et al. (2018b) addressed JavaScript↔CoffeeScript translation with a tree-to-tree approach, which required a strong alignment between the source and target trees.

Chapter 6

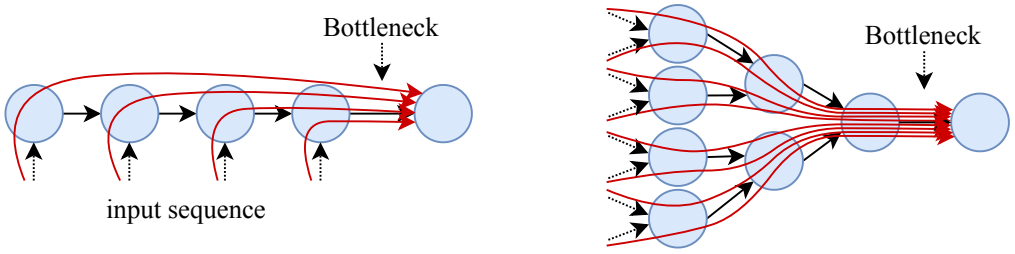
Limitations of Graph Neural Networks

While studying different representations of code, we noticed that graph neural networks, although very versatile and popular, fail to learn long-range patterns in the training data. When trained on programming tasks that *depend* on long-range interactions, we found that GNNs usually *overfit* short-range artifacts in the data. This phenomenon was surprising, because AST paths had no problem learning long-range signals. Searching through the literature, it turned out that since the proposal of the GNNs (Gori et al., 2005; Scarselli et al., 2008), their struggle to propagate information between distant nodes in the graph was one of the major problems in training GNNs. In Section 6.1 we propose a new explanation for this problem: GNNs are susceptible to a *bottleneck* when aggregating messages across a long path. This bottleneck causes the *over-squashing* of exponentially growing information into fixed-size vectors.

Further, we found that Graph Attention Networks (GATs), which are one of the most popular GNN architectures and are considered as the state-of-the-art architecture for representation learning with graphs, can only compute a restricted kind of attention where the ranking of attended nodes is *unconditioned on the query node*. In Section 6.2, we formally define this restricted kind of attention as *static* attention and distinguish it from a strictly more expressive *dynamic* attention. To remove this limitation, we introduce a simple fix by modifying the order of operations and propose GATv2: a *dynamic* graph attention variant that is strictly more expressive than GAT.

6.1 The Bottleneck of Graph Neural Networks and the Over-squashing Phenomenon

Graph neural networks (GNNs) (Gori et al., 2005; Scarselli et al., 2008; Micheli, 2009) have seen sharply growing popularity over the last few years (Duvenaud et al., 2015; Hamilton et al., 2017; Xu et al., 2019). GNNs provide a general framework to model



(a) The bottleneck of RNN seq2seq models (b) The bottleneck of graph neural networks

Figure 6.1: The bottleneck that existed in RNN seq2seq models (before attention) is strictly more harmful in GNNs: information from a node's exponentially-growing receptive field is compressed into a fixed-size vector. Black arrows are graph edges; red curved arrows illustrate information flow.

complex structural data containing elements (nodes) with relationships (edges) between them. A variety of real-world domains such as social networks, computer programs, chemical and biological systems can be naturally represented as graphs. Thus, many graph-structured domains are commonly modeled using GNNs.

A GNN layer can be viewed as a message-passing step (Gilmer et al., 2017), where each node updates its state by aggregating messages flowing from its direct neighbors. GNN variants (Li et al., 2016; Veličković et al., 2018; Kipf and Welling, 2017) mostly differ in how each node aggregates the representations of its neighbors with its own representation. However, most problems also require the interaction between nodes that are not directly connected, and they achieve this by stacking multiple GNN layers. Different learning problems require different ranges of interaction between nodes in the graph to be solved. We call this required range of interaction between nodes – the *problem radius*.

In practice, GNNs were observed *not* to benefit from more than few layers. The accepted explanation for this phenomenon is *over-smoothing*: node representations become indistinguishable when the number of layers increases (Wu et al., 2020). Nonetheless, over-smoothing was mostly demonstrated in *short-range* tasks (Li et al., 2018; Klicpera et al., 2018; Chen et al., 2020a; Oono and Suzuki, 2020; Zhao and Akoglu, 2020; Rong et al., 2020; Chen et al., 2020b) – tasks that have small *problem radii*, where a node's correct prediction mostly depends on its local neighborhood. Such tasks include paper subject classification (Sen et al., 2008) and product category classification (Shchur et al., 2018). Since the learning problems depend mostly on short-range information in these datasets, it makes sense why more layers than the problem radius might be extraneous. In contrast, in tasks that also depend on *long-range* information (and thus have larger *problem radii*), we hypothesize that the explanation for limited performance is *over-squashing*.

To allow a node to receive information from other nodes at a radius of K, the GNN needs to have at least K layers, or otherwise, it will suffer from *under-reaching* – these

distant nodes will simply not be aware of each other. Clearly, to avoid under-reaching, problems that depend on long-range interaction require as many GNN layers as the range of the interaction. However, as the number of layers increases, the number of nodes in each node's receptive field grows *exponentially*. This causes *over-squashing*: information from the exponentially-growing receptive field is compressed into fixed-length node vectors. Consequently, the graph fails to propagate messages flowing from distant nodes, and learns only short-range signals from the training data.

In fact, the GNN bottleneck is analogous to the bottleneck of sequential RNN models. Traditional seq2seq models (Sutskever et al., 2014; Cho et al., 2014a,b) suffered from a bottleneck at every decoder state – the model had to encapsulate the entire input sequence into a fixed-size vector. In RNNs, the receptive field of a node grows *linearly* with the number of recursive applications. However in GNNs, the bottleneck is asymptotically more harmful, because the receptive field of a node grows *exponentially*. This difference is illustrated in Figure 6.1.

Our main contribution in this work is introducing the *over-squashing* phenomenon – a novel explanation for the major and well-known issue of training GNNs for long-range problems, and showing its harmful practical implications. We use a controlled problem to demonstrate how over-squashing prevents GNNs from fitting long-range patterns in the data, and to provide theoretical lower bounds for the required hidden size given the problem radius. We show, analytically and empirically, that GCN (Kipf and Welling, 2017) and GIN (Xu et al., 2019) are susceptible to over-squashing *more* than other types of GNNs such as GAT (Veličković et al., 2018) and GGNN (Li et al., 2016). We further show that prior work that extensively tuned GNNs to real-world datasets suffer from over-squashing: breaking the bottleneck using a simple fully adjacent layer reduces the error rate by 42% in the QM9 dataset, by 12% in ENZYMES, by 4.8% in NCI1, and improves accuracy in VarMisuse, without any additional tuning.

6.1.1 Preliminaries

A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ contains nodes $\mathcal{V}$ and edges $\mathcal{E}$, where $(u, v) \in \mathcal{E}$ denotes an edge from a node u to a node v. For brevity, in the following definitions we treat all edges as having the same *type*; in general, every edge can have a type and features (Schlichtkrull et al., 2018).

Graph neural networks Graph neural networks operate by propagating neural messages between neighboring nodes. At every propagation step (a graph layer): the network computes each node's sent message; every node aggregates its received messages; and each node updates its representation by combining the aggregated incoming messages with its own previous representation.

Formally, each node is associated with an initial representation $\mathbf{h}_v^{(0)} \in \mathcal{R}^{d_0}$. This representation is usually derived from the node's label or its given features. Then, a

GNN layer updates each node's representation given its neighbors, yielding $\mathbf{h}_v^{(1)} \in \mathcal{R}^d$. In general, the k-th layer of a GNN is a parametric function f_k that is applied to each node by considering its neighbors:

$$\mathbf{h}_v^{(k)} = f_k\left(\mathbf{h}_v^{(k-1)}, \{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}_v\}; \theta_k\right) \tag{6.1}$$

where $\mathcal{N}_v$ is the set of nodes that have edges to v: $\mathcal{N}_v = \{u \in \mathcal{V} \mid (u, v) \in \mathcal{E}\}$. The total number of layers K is usually determined empirically as a hyperparameter.

The design of the function f is what mostly distinguishes one type of GNN from the other. For example, graph convolutional networks (GCN) define f as:

$$\mathbf{h}_v^{(k)} = \sigma\left(\sum\nolimits_{u \in \mathcal{N}_v \cup \{v\}} \frac{1}{c_{u,v}} W^{(k)} \mathbf{h}_u^{(k-1)}\right) \tag{6.2}$$

where σ is a nonlinearity such as $ReLU$, and $c_{u,v}$ is a normalization factor often set to $\sqrt{|\mathcal{N}_v| \cdot |\mathcal{N}_u|}$ or $|\mathcal{N}_v|$ (Hamilton et al., 2017). Usually, the last (K-th) layer's output is used for prediction: in node-prediction, $\mathbf{h}_v^{(K)}$ is used to predict a label for v; in graph-prediction, a permutation-invariant "readout" function aggregates the nodes of the final layer using summation, averaging, or a weighted sum (Li et al., 2016).

6.1.2 The GNN Bottleneck

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a given node v, we denote the problem's required range of interaction, the *problem radius*, by r. r is generally unknown in advance, and usually approximated empirically by tuning the number of layers K. We denote the set of nodes in the receptive field of v by $\mathcal{N}_v^K$, which is defined recursively as $\mathcal{N}_v^1 := \mathcal{N}_v$ and $\mathcal{N}_v^K := \mathcal{N}_v^{K-1} \cup \{w \mid (w, u) \in \mathcal{E} \wedge u \in \mathcal{N}_v^{K-1}\}$.

When a prediction problem relies on long-range interaction between nodes, the GNN must have as many layers K as the estimated range of these interactions, or otherwise, these distant nodes would not be able to interact. It is thus required that $K \geq r$. However, the number of nodes in each node's receptive field grows *exponentially* with the number of layers: $\|\mathcal{N}_v^K\| = \mathcal{O}(\exp(K))$ (Chen et al., 2018a). As a result, an exponentially-growing amount of information is squashed into a fixed-length vector (the vector resulting from the $\sum$ in Equation (6.2)), and crucial messages fail to reach their distant destinations. Instead, the model learns only short-ranged signals from the training data and consequently might generalize poorly at test time.

Example Consider the NeighborsMatch problem of Figure 6.2. Green nodes (A, B, C) have a varying number of blue neighbors ( ) and an alphabetical label. Each example in the dataset is a different graph that has a different mapping from numbers of neighbors to labels. The rest of the graph (marked as ) represents a general, unknown, graph structure. The goal is to predict a label for the target node, which is marked with a question mark (?), according to its number of blue neighbors. The

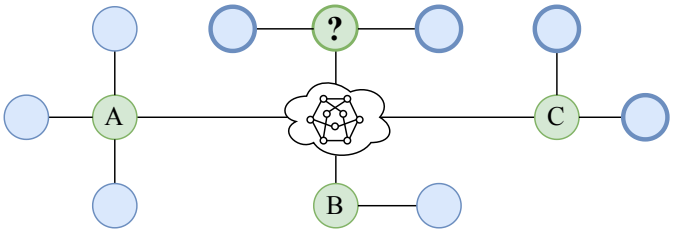


Figure 6.2: The NeighborsMatch problem: green nodes have blue neighbors and an alphabetical label. The goal is to predict the label (A, B, or C) of the green node that has the same number of blue neighbors as the target node in the same graph. In this example, the correct label is **C**, because the target node has *two* blue neighbors, like the node marked with C in the same graph.

correct answer is **C** in this case, because the target node has *two* blue neighbors, like the node marked with C in the same graph. Every example in the dataset has a different mapping from numbers of neighbors to labels, and thus message propagation and matching between the target node and all the green nodes must be performed *for every graph in the dataset*.

Since the model must propagate information from *all* green nodes before predicting the label, a bottleneck at the target node is inevitable. This bottleneck causes *over-squashing*, which can prevent the model from fitting the training data perfectly. provide theoretical lower bounds for the GNN's hidden size. Obviously, adding direct edges between the target node and the green nodes, or making the existing edges bidirectional, could ease information flow for this specific problem. However, in real-life domains (e.g., molecules), we do not know the optimal message propagation structure a priori, and must use the given relations (such as bonds between atoms) as the graph's edges.

Although this is a contrived problem, it resembles real-world problems that are often modeled as graphs. For example, a computer program in a language such as Python may declare multiple variables (i.e., the green nodes in Figure 6.2) along with their types and values (their numbers of blue neighbors in Figure 6.2); later in the program, predicting which variable should be used in a specific location (predict the alphabetical label in Figure 6.2) must use one of the variables that are available in scope based on the required type and the required value at that point.

Short- vs. long-range problems Much of prior GNN work has focused on problems that were local in nature, with small problem radii, where the underlying inductive bias was that a node's most relevant context is its local neighborhood, and long-range interaction was not necessarily needed. With the growing popularity of GNNs, their adoption expanded to domains that required longer-range information propagation as well, without addressing the inherent bottleneck. In this work, we focus on problems that *require* long-range information. That is, a correct prediction requires considering the local environment of a node *and* interactions beyond the close neighborhood. For example, a chemical property of a molecule (Ramakrishnan et al., 2014; Gilmer et al.,

2017) can depend on the combination of atoms that reside in the molecule's *opposite sides*. Problems of this kind require long-range interaction, and thus, a large number of GNN layers. Since the receptive field of each node grows exponentially with the number of layers, the more layers – over-squashing is more harmful.

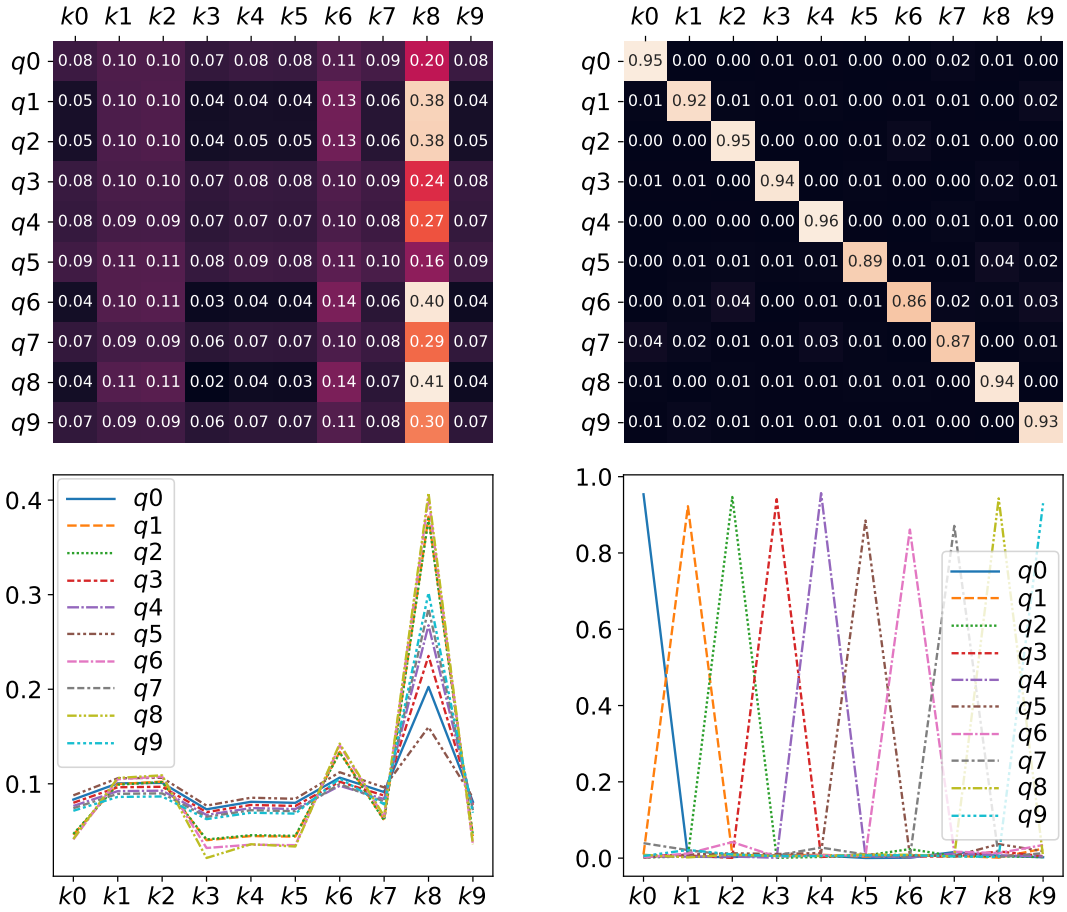
In problems that are local in nature (small r) – the bottleneck is less troublesome, because a GNN can perform well with only few layers (e.g., K=2 layers in Kipf and Welling (2017)), and the receptive field of a node can be exponentially smaller. Domains such as citation networks (Sen et al., 2008), social networks (Leskovec and Mcauley, 2012), and product recommendations (Shchur et al., 2018) usually raise short-range problems and are thus *not* the focus of this work. So, how long is long-range? We discuss and analyze this question theoretically in Alon and Yahav (2021).

Evaluation In Alon and Yahav (2021), we show empirical evaluation that demonstrates that the GNN bottleneck exists and raises over-squashing starting from values of r as small as $r = 4$: we generated a synthetic benchmark that is theoretically solvable; however, in practice, all GNNs fail to reach 100% training accuracy because of the bottleneck. Second, we find that the bottleneck exists in prior work, which addressed real-world problems, by showing that the original implementations of the authors can be further improved by considering the bottleneck. Finally, we find that GNNs that absorb incoming edges equally, like GCN (Kipf and Welling, 2017) and GIN (Xu et al., 2019), are more susceptible to over-squashing than GNNs that use attention to weigh incoming edges like GAT (Veličković et al., 2018) and GGNN (Li et al., 2016).

More details can be found in Alon and Yahav (2021).

6.2 How Attentive are Graph Attention Networks?

Graph Attention Networks (GATs) are one of the most popular GNN architectures and are considered as the state-of-the-art architecture for representation learning with graphs. In GAT, every node attends to its neighbors given its own representation as the query. However, in this work we show that GATs can only compute a restricted kind of attention where the ranking of attended nodes is *unconditioned on the query node*. We formally define this restricted kind of attention as *static* attention and distinguish it from a strictly more expressive *dynamic* attention. Because GATs use a *static* attention mechanism, there are simple graph problems that GAT cannot express: in a controlled problem, we show that static attention hinders GAT from even fitting the training data. To remove this limitation, we introduce a simple fix by modifying the order of operations and propose GATv2: a *dynamic* graph attention variant that is strictly more expressive than GAT. We perform an extensive evaluation and show that GATv2 outperforms GAT across 11 OGB and other benchmarks while we match their parametric costs. Our code is available at `https://github.com/tech-srl/how_attentive_are_gats` .



(a) Attention in standard GAT (Veličković et al. (2018))

(b) Attention in GATv2, our fixed version of GAT

Figure 6.3: Standard GAT (Figure 6.3a) computes *static* attention: the ranking of attention coefficients is global for all nodes in the graph, and is unconditioned on the query node. For example, all queries ($q0$ to $q9$) attend mostly to the 8th key ($k8$). In contrast, GATv2 (Figure 6.3b) can actually compute *dynamic* attention, where every query has a different ranking of attention coefficients of the keys.

Graph neural networks (GNNs; Gori et al., 2005; Scarselli et al., 2008) have seen increasing popularity over the past few years (Duvenaud et al., 2015; Atwood and Towsley, 2016; Bronstein et al., 2017; Monti et al., 2017). GNNs provide a general and efficient framework to learn from graph-structured data. Thus, GNNs are easily applicable in domains where the data can be represented as a set of nodes and the prediction depends on the relationships (edges) between the nodes. Such domains include molecules, social networks, product recommendation, computer programs and more.

A GNN can be viewed as a message-passing network (Gilmer et al., 2017), where each node iteratively updates its state by interacting with its neighbors. GNN variants (Wu et al., 2019; Xu et al., 2019; Li et al., 2016) mostly differ in how each node aggregates the representations of its neighbors and combines them with its own repre-

sentation. Veličković et al. (2018) pioneered the use of attention-based neighborhood aggregation, in one of the most popular GNN variants – Graph Attention Network (GAT). In GAT, every node updates its representation by attending to its neighbors using its own representation as the query. This generalizes the standard averaging or max-pooling of neighbors (Kipf and Welling, 2017; Hamilton et al., 2017), by allowing every node to compute a *weighted* average of its neighbors. The work of Veličković et al. also generalizes the Transformer's (Vaswani et al., 2017) self-attention mechanism, from sequences to graphs (Joshi, 2020).

While GAT is one of the most popular GNN architectures (Bronstein et al., 2021) and is considered as the state-of-the-art neural architecture for learning with graphs (Wang et al., 2019a), we show that *GATs do not actually compute dynamic attention*, a fact that severely hinders their expressiveness. Instead, we show that GAT only uses a restricted "static" form of attention: for every query node, attention is *monotonic* with respect to its neighbor key scores. That is, the ranking (the $argsort$) of attention coefficients is shared across all nodes in the graph, and is unconditioned on the query node. This limitation of the standard GAT is demonstrated in Figure 6.3a.

Supposedly, the conceptual idea of attention as the form of interaction between GNN nodes is orthogonal to the specific choice of attention function. However, Veličković et al.'s original design of GAT has spread to a variety of domains (Wang et al., 2019a; Qiu et al., 2018; Yang et al., 2020; Wang et al., 2019c; Huang and Carley, 2019; Ma et al., 2020; Kosaraju et al., 2019; Nathani et al., 2019; Wu et al., 2020; Zhang et al., 2020) and has become the default implementation of "graph attention network" in all popular GNN libraries such as PyTorch Geometric (Fey and Lenssen, 2019), DGL (Wang et al., 2019b), and others (Dwivedi et al., 2020; Gordić, 2020; Brockschmidt, 2020).

To overcome the limitation we identified in GAT, we introduce a simple fix to its attention function by modifying the order of internal operations. The result is GATv2 – a graph attention variant that has a universal approximator attention function, and is thus *strictly more expressive than GAT*. The effect of fixing the attention function in GATv2 is demonstrated in Figure 6.3b.

In summary, our main contribution is identifying that one of the most popular GNN types, the graph attention network, cannot actually compute dynamic attention. We introduce formal definitions for analyzing the expressive power of graph attention mechanisms, and derive our claims theoretically from the equations of Veličković et al. (2018). Empirically, we use a synthetic problem to show that standard GAT *cannot express* alignment problems that require *dynamic* attention. We introduce a simple fix by switching the order of internal operations in the attention function of GAT, and propose GATv2, which does compute dynamic attention. We further conduct a thorough empirical comparison of GAT and GATv2 and find that GATv2 outperforms GAT across 11 benchmarks of node-, link-, and graph-prediction. For example, GATv2 outperforms extensively tuned GNNs by over 1.4% in the difficult "UnseenProj Test" set

of the VarMisuse task (Allamanis et al., 2018), without any hyperparameter tuning; and GATv2 improves over an extensively-tuned GAT by 11.5% in 13 prediction objectives in QM9. In node-prediction benchmarks from OGB (Hu et al., 2020), not only that GATv2 outperforms GAT with respect to accuracy – we find that GATv2 is also much more robust to noise.

More details can be found in Brody et al. (2021).

This work was loosely inspired by my previous experience in examining attention in speech recognition models. More details about this can be found in Alon et al. (2019b).

Chapter 7

Additional Applications

This section surveys additional applications of programming language processing.

Edit Completion We address the problem of predicting *edit completions* based on a learned model that was trained on past edits. Given a code snippet that is partially edited, our goal is to predict a *completion of the edit for the rest of the snippet*. We refer to this task as the EditCompletion task and present a novel approach for tackling it. The main idea is to directly represent structural edits. This allows us to model the likelihood of the edit itself, rather than learning the likelihood of the edited code. We represent an edit operation as a path in the program's Abstract Syntax Tree (AST), originating from the source of the edit to the target of the edit. Using this representation, we present a powerful and lightweight neural model for the EditCompletion task.

We conduct a thorough evaluation, comparing our approach to a variety of representation and modeling approaches that are driven by multiple strong models such as LSTMs, Transformers, and neural CRFs. Our experiments show that our model achieves a 28% relative gain over state-of-the-art sequential models and 2× higher accuracy than syntactic models that learn to generate the edited *code*, as opposed to modeling the *edits* directly.

Our code, dataset, and trained models are publicly available at `https://github.com/tech-srl/c3po/` .

Adversarial Examples Neural models of code have shown impressive results when performing tasks such as predicting method names and identifying certain kinds of bugs. We show that these models are vulnerable to *adversarial examples*, and introduce a novel approach for *attacking* trained models of code using adversarial examples. The main idea of our approach is to force a given trained model to make an incorrect prediction, as specified by the adversary, by introducing small perturbations that do not change the program's semantics, thereby creating an adversarial example. To find such perturbations, we present a new technique for Discrete Adversarial Manipulation

of Programs (DAMP). DAMP works by deriving the desired prediction with respect to the model's *inputs*, while holding the model weights constant, and following the gradients to slightly modify the input code.

We show that our DAMP attack is effective across three neural architectures: CODE2VEC, GGNN, and GNN-FILM, in both Java and C#. Our evaluations demonstrate that DAMP has up to 89% success rate in changing a prediction to the adversary's choice (a targeted attack) and a success rate of up to 94% in changing a given prediction to any incorrect prediction (a non-targeted attack). To defend a model against such attacks, we empirically examine a variety of possible defenses and discuss their trade-offs. We show that some of these defenses can dramatically drop the success rate of the attacker, with a minor penalty of 2% relative degradation in accuracy when they are not performing under attack.

Our code, data, and trained models are available at `https://github.com/tech-srl/adversarial-examples` .

Neural Reverse Engineering In David et al. (2020), we address the problem of reverse engineering of stripped executables, which contain no debug information. This is a challenging problem because of the low amount of syntactic information available in stripped executables, and the diverse assembly code patterns arising from compiler optimizations.

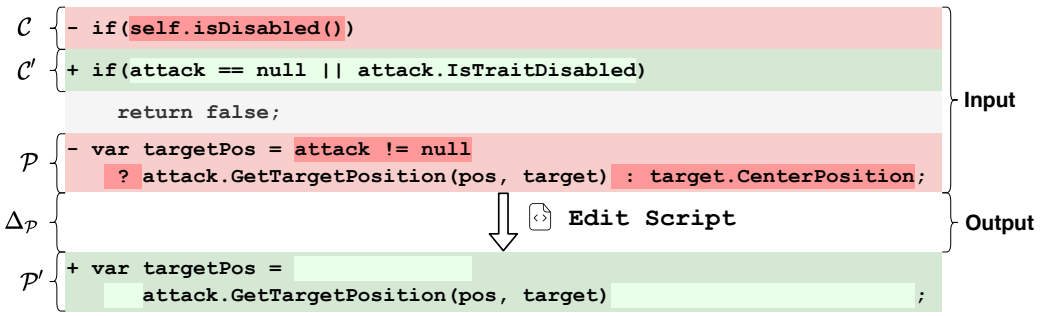
We present a novel approach for predicting procedure names in stripped executables. Our approach combines static analysis with neural models. The main idea is to use static analysis to obtain augmented representations of *call sites*; encode the structure of these call sites using the control-flow graph (CFG) and finally, generate a target name while attending to these call sites. We use our representation to drive graph-based, LSTM-based and Transformer-based architectures.

Our evaluation shows that our models produce predictions that are difficult and time consuming for humans, while improving on existing methods by 28% and by 100% over state-of-the-art neural textual models that do not use any static analysis. Code and data for this evaluation are available at `https://github.com/tech-srl/Nero` .

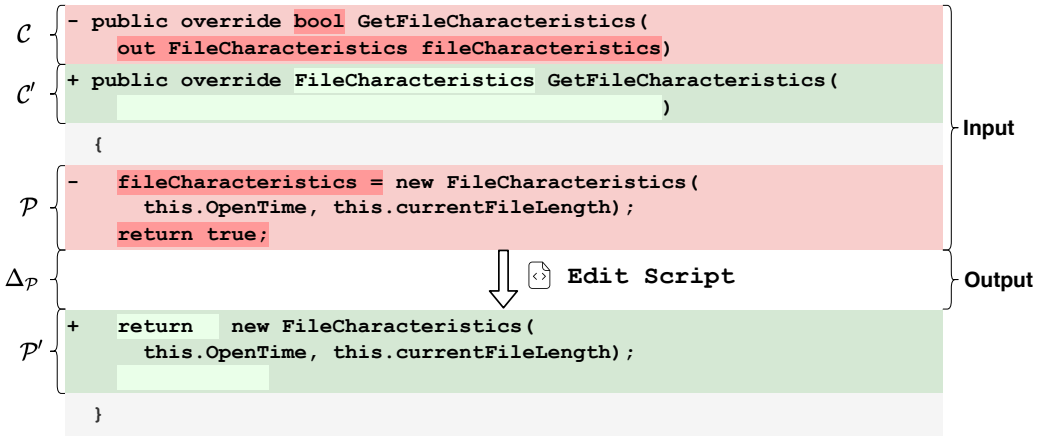
7.1 Neural Edit Completion

Software development is an evolutionary process. Programs are being maintained, refactored, fixed, and updated on a continuous basis. Program *edits* are therefore at the very core of software development. Poor edits can lead to bugs, security vulnerability, unreadable code, unexpected behavior, and more. The ability to suggest a good edit in code is therefore crucial.

We introduce the EDITCOMPLETION task: predict *edit completions* based on a learned model that was trained on past edits. Given a code snippet that is partially edited, our goal is to predict an *edit completion* that *completes the edit for the rest of the snippet.*

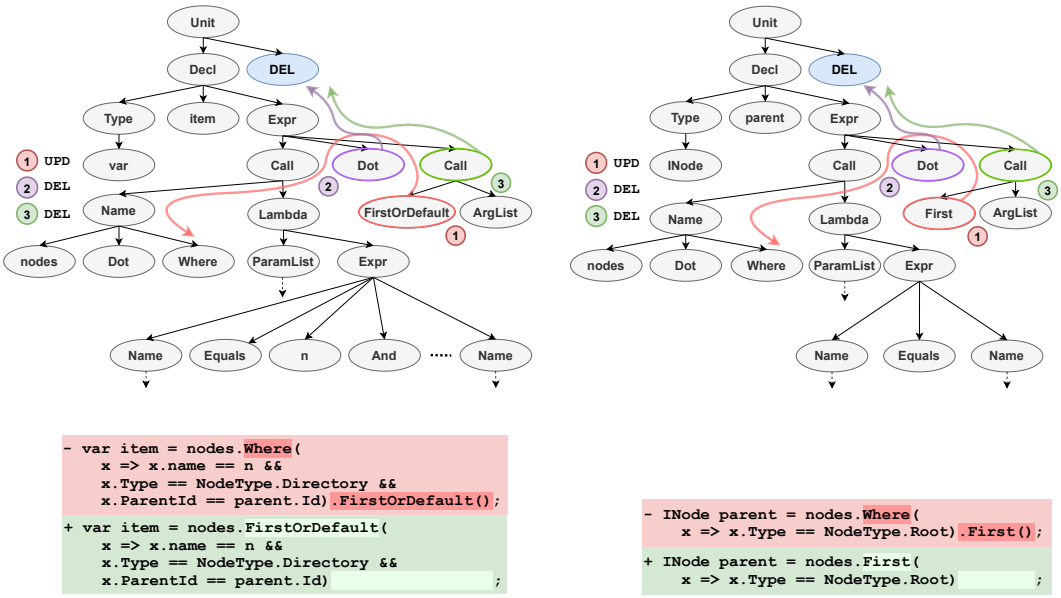


(a) The predicate of the `if` statement in $\mathcal{C}$ was edited to include a `null` check for `attack`. Thus, in $\mathcal{P}$, the checking of `attack != null` and the ternary operator can be removed.



(b) The signature of `GetFileCharacteristics` in $\mathcal{C}$ was edited to return a `FileCharacteristic` object instead of modifying an output parameter. Thus, in $\mathcal{P}$, the method should return a `FileCharacteristic` object instead of returning `true`.

Figure 7.1: Examples of EditCompletion. The input consists of a program fragment $\mathcal{P}$ and edits that occurred in the context that transformed $\mathcal{C}$ into $\mathcal{C}'$. The output is $\Delta_{\mathcal{P}}$ – an edit script that describes the likely edit. Applying $\Delta_{\mathcal{P}}$ to $\mathcal{P}$ results in $\mathcal{P}'$ – the code after the edit.




```
- var item = nodes.Where(
    x => x.name == n &&
    x.Type == NodeType.Directory &&
    x.ParentId == parent.Id).FirstOrDefault();
+ var item = nodes.FirstOrDefault(
    x => x.name == n &&
    x.Type == NodeType.Directory &&
    x.ParentId == parent.Id)                    ;
```

(a)

```
- INode parent = nodes.Where(
    x => x.Type == NodeType.Root).First();
+ INode parent = nodes.First(
    x => x.Type == NodeType.Root)              ;
```

(b)

Figure 7.2: An example of two edits. These examples are different and the edits operate on different values. However, observing the structure of these edits reveals the similarity between them and allows a learning model to generalize better. This similarity is expressed as almost identical AST paths. For simplicity, only the program fragment that should be edited $\mathcal{P}$ is shown, without the context $\mathcal{C}$.

The edit completion is represented technically as a sequence of edit operations that we refer to as an *edit script*.

Problem Definition Let $\mathcal{P}$ be a given program fragment and $\mathcal{C}$ be the surrounding context of $\mathcal{P}$ before any edits were applied. Let $\Delta_{\mathcal{C}}$ denote the edits that were applied to $\mathcal{C}$, and $\mathcal{C}' = \Delta_{\mathcal{C}}(\mathcal{C})$ the resulting edited context. The goal in our EditCompletion task is to predict an edit function $\Delta_{\mathcal{P}}$, such that applying $\Delta_{\mathcal{P}}$ to $\mathcal{P}$ results in the program fragment after the edit: $\Delta_{\mathcal{P}}(\mathcal{P}) = \mathcal{P}'$. Our underlying assumption is that the distribution of edits in $\mathcal{P}$ can be inferred from the edits $\Delta_{\mathcal{C}}$ that occurred in its context. We thus model the probability: $Pr(\Delta_{\mathcal{P}} \mid \Delta_{\mathcal{C}})$. We present a new approach for representing and predicting $\Delta_{\mathcal{P}}$ in the EditCompletion task, named C^3: Contextual Code Changes.

Motivating Examples Consider the EditCompletion examples in Figure 7.1a and Figure 7.1b. These illustrate the significance of edits in the context $\mathcal{C}$ and how they can help in suggesting a likely edit for $\mathcal{P}$. In Figure 7.1a, the edit in the context consists of changing the `if` statement predicate, resulting in a `null` check for the variable `attack`. After the edit in the context, the value of `attack` in $\mathcal{P}$ cannot be `null`. Therefore, the ternary statement that checks `attack` for nullness in $\mathcal{P}$ can be removed. Our model successfully predicted the needed edit $\Delta_{\mathcal{P}}$, which is applied to $\mathcal{P}$ to yield $\mathcal{P}'$.

Figure 7.1b shows another example, in which the edit in the context is a modification of a function signature. In $\mathcal{C}'$, the return type was changed to `FileCharacteristics`, and the output parameter `fileCharacteristics` for the function was removed. $\mathcal{P}$ consists of an assignment to the parameter `fileCharacteristics`, and a return statement of `true` value. The edit in the context implies a necessary edit in $\mathcal{P}$, in which the assignment statement has to be removed (since `fileCharacteristics` is no longer defined) and the return statement must include a variable of type `FileCharacteristics`. Our model successfully predicted the correct edit for $\mathcal{P}$. $\mathcal{P}'$ consists of returning an object of type `FileCharacteristics`.

Edit Completion vs. Code Completion It is important to note that EditCompletion and code completion are completely different tasks. The goal of code completion is to predict missing fragments of a program, given a partial program as context. In contrast, the goal of EditCompletion is to predict additional edits in a partial sequence of edit operations. That is, while code completion operates on code, EditCompletion operates on *code edits*.

Representing Code Edits The main design decision in learning code edits is *how to represent the edit*, i.e., how to represent the difference between the code in its original form and its desired, altered, form. Naïvely, differencing programs can be performed by treating the code as text and using *text-diff* algorithms for line- or inline-differencing. In contrast, we model the *difference between the abstract syntax trees* (ASTs) of the original and the edited code. This allows us to naturally use paths in the AST (AST paths) to model edits.

Our Approach We present a novel approach for EditCompletion: predicting contextual code changes – C^3. Code changes can be described as a sequence of edit operations, such as *"move a node, along with its underlying subtree, to be a child of another node"* or *"update the value of a node to be identical to the value of another node"*. Such edit operations can be naturally represented as paths between the source node and the target node, along with the relationship between them and the edit command, i.e., "move" or "update". AST paths provide a natural way to express binary relationships between nodes (and thus subtrees) in the AST. We use AST paths to represent $\Delta_{\mathcal{C}}$ – edits that occurred in the context and transformed $\mathcal{C}$ into $\mathcal{C}'$, such that $\Delta_{\mathcal{C}}(\mathcal{C}) = \mathcal{C}'$. We also use AST paths to represent $\Delta_{\mathcal{P}}$ – the edits that should be applied to $\mathcal{P}$. We thus model the probability $Pr(\Delta_{\mathcal{P}} \mid \Delta_{\mathcal{C}})$, where both the input $\Delta_{\mathcal{C}}$ and the output $\Delta_{\mathcal{P}}$ are represented as AST paths.

Representing edits as paths allows a learning model to generalize well across different examples. Consider the two examples in Figure 7.2. In Figure 7.2a, the edit modifies a series of LINQ calls, converting `Where(<predicate>).FirstOrDefault()` into `FirstOrDefault(<predicate>)`. The edit in Figure 7.2b modifies `Where(<predicate>).First()`

into `First(<predicate>)`. Although the predicates are different and these edits operate on different values, the structure of the edits in Figure 7.2a and Figure 7.2b is identical. This similarity is expressed in the AST paths that represent these edits. For example, consider the identical structure of the path ① in the two figures, where it operates on a different value in each figure (`FirstOrDefault` and `First`).

Our use of AST paths allows the model to generalize these edits, even though these edits are not identical and their predicates are different.

We apply a Pointer Network (Vinyals et al., 2015) to point to paths in the AST of $\mathcal{P}$ and create an edit operation sequence, i.e., an edit script. While prior work used AST paths to *read* programs and predict a label Chapters 3 and 4, we generate an edit script by *predicting* AST paths, i.e., making AST paths *the output* of our model.

We show the effectiveness of C^3 on EditCompletion on a new dataset, scraped from over 300,000 commits in GitHub.

Our approach significantly outperforms textual and syntactic approaches that either model the code or model only the edit, and are driven by strong neural models.

Contributions The main contributions of this work are:

- We introduce the EditCompletion task: given a program $\mathcal{P}$ and edits that occurred in its context, predict the likely edits that should be applied to $\mathcal{P}$.
- C^3 – a novel approach for representing and predicting contextual edits in code. This is the first approach that represents structural edits directly.
- Our technique directly captures the relationships between subtrees that are changed in an edit using *paths* in the AST. The output of our technique is an edit script that is executed to edit the program fragment $\mathcal{P}$.
- A prototype implementation of our approach, called C^3PO, for Contextual Code Changes via Path Operations. C^3PO is implemented using a strong neural model that predicts the likely edit by pointing to an AST path that reflects that edit.
- A new EditCompletion dataset of source code edits and their surrounding context edits, scraped from over 300,000 commits in GitHub.
- An extensive empirical evaluation that compares our approach to a variety of representation and modeling approaches, driven by strong models such as LSTMs, Transformers, and neural CRFs. Our evaluation shows that our model achieves over 28% relative gain over state-of-the-art strong sequential models, and over 2× higher accuracy than syntactic models that do not model edits directly.
- A thorough ablation study that examines the contribution of syntactic and textual representations in different components of our model.

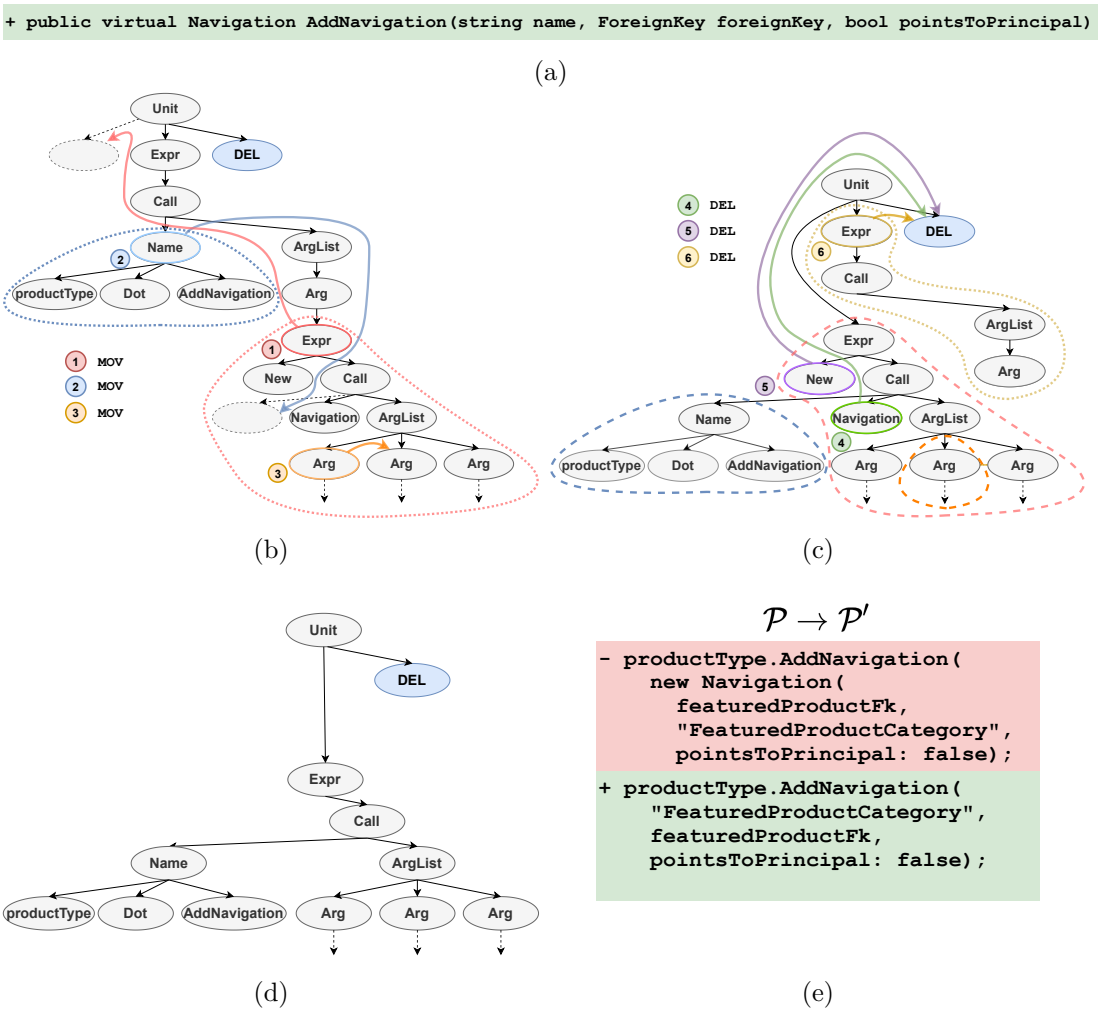


Figure 7.3: An EditCompletion example from our test set. Figure 7.3a shows the edit that transforms $\mathcal{C}$ into $\mathcal{C}'$ – overloading the function `AddNavigation`. Figure 7.3e shows $\mathcal{P}$ and $\mathcal{P}'$ as code in red and green, respectively. Figure 7.3b depicts the partial AST and the first three edit operations of the edit. Figure 7.3c shows the AST after applying the first three operations, and shows the next three operations as AST paths. Figure 7.3d illustrates the AST after performing all operations, resulting in an AST that corresponds to $\mathcal{P}'$. Every edit operation is represented by an AST path having the same color and number as the edit command. Dotted contours represent subtrees that will be affected by applying these operations.

7.1.1 Motivating Example

In this section, we demonstrate our approach using a simple EditCompletion example. The main idea is to represent all valid edit operations in $\mathcal{P}$ as AST paths, and predict a sequence of these paths. Since every path is associated with an edit operation, by pointing to a sequence of paths, we, in fact, predict an edit script.

High-level Overview Consider the edit that occurred in the context of Figure 7.3a – insertion of a new definition of the method `AddNavigation`, which overloads previous

definitions. After applying this edit, it is possible to use this new signature when calling `AddNavigation`. Consider the original code snippet $\mathcal{P}$ at the top of Figure 7.3e. The edit in the context allows us to simplify the call to `AddNavigation` using the new signature, as shown in the "edited" code snippet $\mathcal{P}'$ at the bottom of Figure 7.3e. Consider the partial AST of $\mathcal{P}$ in Figure 7.3b. The desired edit can be described as an edit script consisting of six edit operations to the AST of $\mathcal{P}$. Consider the first operation: ① `MOV`. The meaning of this operation is to move the node `Expr` with its subtree to be the leftmost child of the node `Unit`. This edit operation can be represented by the **red** ① path: `Expr` → `Arg` → `ArgList` → `Call` → `Expr` → `Unit`. Note how this path directly captures the syntactic relationship between the node `Expr` and the node `Unit`, allowing our model to predict a `MOV` operation as part of the edit script.

In Figure 7.3c we can see the result of applying the following first three operations: ① `MOV`, ② `MOV`, ③ `MOV`, moving subtrees to new locations in the tree. The last three commands are `DEL` operations, expressing deletion of a node and its underlying subtree. These operations can be represented using paths as well. For instance, ④ `DEL` is represented by the **green** ④ path: `Navigation` → `Call` → `Expr` → `Unit` → `DEL`, where `DEL` is an artificial node that we add as a child of the AST's root. In Figure 7.3d we can see the AST after applying all six operations. After executing all six operations, our model produces $\mathcal{P}'$, shown in Figure 7.3e.

Path Extraction To inform the model about the available edits it can use for prediction, we parse the AST of $\mathcal{P}$ to extract all AST paths that represent valid edits. Every path can represent different edit "commands" that use the same path. For example, consider the **blue** ② path in Figure 7.3b: `Name` → `Call` → `ArgList` → `Arg` → `Expr` → `Call`. This path can represent a move operation – `MOV`, i.e., moving the node `Name` with its subtree, to be the leftmost child of `Call`; alternatively, this path can represent an insertion operation – `INS`, i.e., copy `Name` with its subtree, and insert it as the leftmost child of `Call`. To distinguish between different edit operations that are represented using the same AST path, each path is encoded as a vector once, and projected into three vectors using different learned functions. Each resulting vector corresponds to a different kind of edit operation. For example, the **orange** ③ path in Figure 7.3b can represent either "move" (`MOV`), "update" (`UPD`) or "insert" (`INS`) operations. In this case, this path was projected using the learned function that represents "move".

Edit Script Prediction We predict one edit operation at each step by *pointing* at a path and its associated operation from among the valid edit operations. This results in an *edit script*. For example, in Figure 7.3, our model finds that the **red** ① path with `MOV` is most likely to be the first operation. Then, given this edit, our model finds that the **blue** ② path with `MOV` is most likely to be the next operation, and so on, until we predict a special "end of sequence" (`EOS`) symbol.

Modeling Code Likelihood vs. Modeling Edit Likelihood Modeling edits using AST paths provides an effective way to model *only the difference* between $\mathcal{P}$ and $\mathcal{P}'$. For example, consider the **red** ① path that moves the subtree rooted at `Expr` from its original place to be the first child of `Unit`. To predict this edit, our model only needs to select the **red** ① path out of the other available operations. In contrast, a model that attempts to generate $\mathcal{P}'$ entirely (Chen et al., 2019), would need to generate the entire subtree from scratch in the new location.

Pairwise Edit Operations Most edit operations, such as "move" and "update", can be described as *pairwise* operations, having the "source" and the "target" locations as their two arguments. AST paths provide a natural way to represent pairwise relations, originating from the "source" location, and reaching the "target" location through the shortest path between them in the tree. In contrast, prior work that used only *unary* edit operations such as HOPPITY (Dinella et al., 2020) are limited to inserting each node individually, and thus use *multiple* edit commands to express the ① `MOV` operation. Our model represents this edit operation as a single AST path – the **red** ① path.

Key aspects The example in Figure 7.3 demonstrates several key aspects of our method:

- Edits applied to the context of $\mathcal{P}$ can provide useful information for the required edit to $\mathcal{P}$.
- Pairwise edit operations can be naturally represented as AST paths.
- A neural model, trained on these paths, can generalize well to other programs, thanks to the direct modeling of code edits as paths.
- By *pointing* at the available edit operations, the task that the model addresses becomes *choosing* the most likely edit, rather than generating $\mathcal{P}'$ from scratch, and thus significantly eases the learning task.

More details can be found in Brody et al. (2020).

7.2 Adversarial Examples for Models of Code

In domains such as computer vision, deep models have been shown to be vulnerable to *adversarial examples* (Szegedy et al., 2013; Goodfellow et al., 2014b). Adversarial examples are inputs crafted by an adversary to force a trained neural model to make a certain (incorrect) prediction. The generation of adversarial examples was demonstrated for image classification (Szegedy et al., 2013; Goodfellow et al., 2014b,a) and for other domains (Carlini and Wagner, 2018; Taori et al., 2019; Ebrahimi et al., 2017; Pruthi et al., 2019). The basic idea underlying many of the techniques is to add

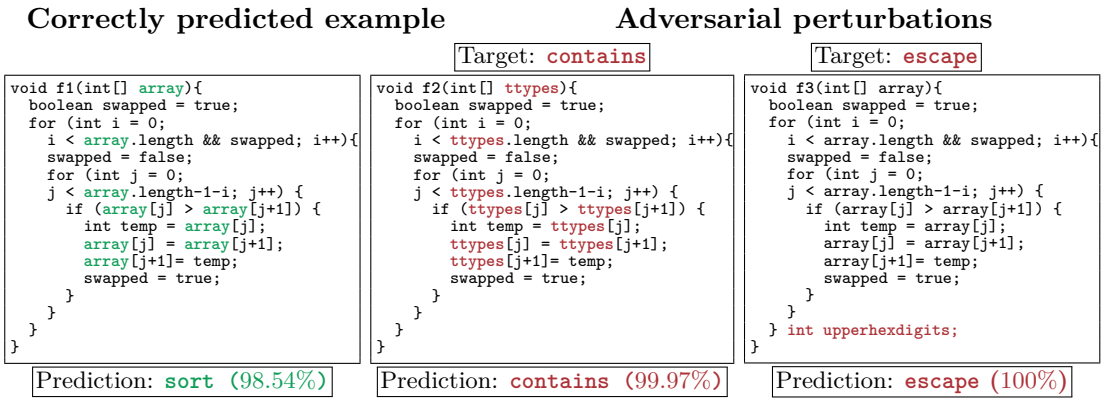


Figure 7.4: A Java snippet `f1` is classified correctly as `sort` by the model of `code2vec.org`. Given `f1` and the target `contains`, our approach generates `f2` by renaming `array` to `ttypes`. Given the target `escape`, our approach generates `f3` by adding an unused variable declaration of `int upperhexdigits`. Additional examples can be found in Yefet et al. (2020).

specially-crafted noise to a correctly labeled input, such that the model under attack yields a desired incorrect label when presented with the modified input (i.e., with the addition of noise). Adding noise to a *continuous object* to change the prediction of a model is relatively easy to achieve mathematically. For example, for an image, this can be achieved by changing the intensity of pixel values (Szegedy et al., 2013; Goodfellow et al., 2014b). Unfortunately, this does not carry over to the domain of programs, since a program is a *discrete object* that must maintain semantic properties.

In this work, we present a novel approach for generating adversarial examples for neural models of code. More formally:

7.2.1 Goal

Given a program $\mathcal{P}$ and a correct prediction y made by a model $\mathcal{M}$, such that: $\mathcal{M}(\mathcal{P}) = y$, our goal is to find a semantically equivalent program $\mathcal{P}'$ such that $\mathcal{M}$ makes a given adversarial prediction y_{bad} of the adversary's choice: $\mathcal{M}(\mathcal{P}') = y_{bad}$.

The main challenge in tackling the above goal lies in exploring the vast space of programs that are semantically equivalent to $\mathcal{P}$, and finding a program for which $\mathcal{M}$ will predict y_{bad}.

Generally, we can define a set of semantic-preserving transformations, which in turn induce a space of semantically equivalent programs. For example, we can rename variables; and add dead code.

There are clearly many other semantic preserving transformations (e.g., re-ordering independent statements), but their application would require a deeper analysis of the program to guarantee that they are indeed semantic preserving. In this work, therefore, we focus on the above two semantic-preserving transformations, which can be safely applied without any semantic analysis.

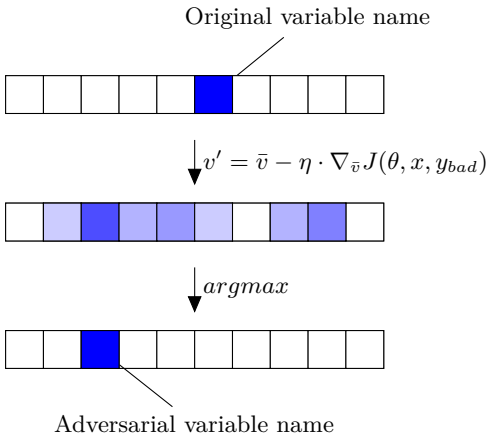


Figure 7.5: Perturbing a variable name: the original variable name is represented as a one-hot vector over the variable-name vocabulary. After perturbation, the vector is no longer one-hot. We apply argmax to find the most likely adversarial name, resulting with another one-hot vector over the variable-name vocabulary.

One naïve approach for exploring the space of equivalent programs is to randomly apply transformations using brute-force. We can apply transformations randomly to generate new programs and use the model to make a prediction for each generated program. However, the program space to be explored is exponentially large, making exhaustive exploration prohibitively expensive.

7.2.2 Our Approach

We present a new technique called Discrete Adversarial Manipulation of Programs (DAMP). The main idea in DAMP is to select semantic preserving perturbations by deriving the output distribution of the model with respect to the model's input and following the gradient to modify the input, while keeping the model weights constant. Given a desired adversarial label y_{bad} and an existing variable name, we derive the loss of the model with y_{bad} as the correct label, with respect to the one-hot vector of the input variable. We then take the argmax of the resulting gradient to select an alternative variable name, rename the original variable to the alternative name, check whether this modification changes the output label to the desired adversarial label, and continue iteratively. This process is illustrated in Figure 7.5.

This iterative process allows DAMP to modify the program in a way that preserves its semantics but will cause a model to make adversarial predictions. We show that models of code are susceptible to *targeted attacks* that force a model to make a specific incorrect prediction chosen by the adversary, as well as to simpler *non-targeted attacks* that force a model to make *any incorrect prediction* without a specific target prediction in mind. Our approach is a "white-box" approach, since it assumes the attacker has access to either the model under attack or to a similar model. [1] Under this assumption,

[1] As recently shown by Wallace et al. (2020), this is a reasonable assumption. An attacker can

our approach is general and *is applicable to any model that can be derived with respect to its inputs* i.e., any neural model. We do not make any assumptions about the internal details or specific architecture of the model under attack.

To mitigate these attacks, we evaluate and compare a variety of *defensive* approaches. Some of these defenses work by re-training the model using another loss function or a modified version of the same dataset. Other defensive approaches are "modular", in the sense that they can be placed in front of an already-trained model, identify perturbations in the input, and feed a masked version of the input into the vulnerable model. These defense mechanisms allow us to trade off the accuracy of the original model for improved robustness.

Main Contributions The contributions of this work are:

- The first technique for generating *targeted* adversarial examples for models of code. Our technique, called Discrete Adversarial Manipulation of Programs (DAMP), is general and only requires that the attacker is able to compute gradients in the model under attack (or in a similar model). DAMP is effective in generating both targeted and non-targeted attacks.
- An experimental evaluation of attacks on three neural architectures: code2vec (Alon et al., 2019c), GGNN (Allamanis et al., 2018), and GNN-FiLM (Brockschmidt et al., 2019) in two languages: Java and C#. Our evaluation shows that our adversarial technique can change a prediction according to the adversary's will ("targeted attack") up to 89% of the time, and is successful in changing a given prediction to an incorrect prediction ("non-targeted attack") 94% of the time.
- A thorough evaluation of techniques for defending models of code against attacks that perturb names, and an analysis of their trade-offs. When some of these defenses are used, the success rate of the attack drops drastically for both targeted and non-targeted attacks, with a minor penalty of 2% in accuracy.

7.2.3 Motivating Examples

We begin by demonstrating our technique on two examples, which address two different tasks, using two different neural models, and in two programming languages (Java and C#).

Bypass Semantic Labeling (code2vec - Java) We demonstrate how our approach can force the CODE2VEC (Alon et al., 2019c) model to predict a label of our choice. Consider the code snippet `f1` of Figure 7.4. This code snippet sorts a given array. The

imitate the model under attack by: training an imitation model using labels achieved by querying the original model; crafting adversarial examples using the imitation model; and transferring these adversarial examples back to the original model.

Correctly predicted example **Adversarial perturbation**

Target: SourceType

```
struct TypePair : IEquatable<TypePair>
{
  public static TypePair Create<TSource,
      TDestination>(TSource source,
        TDestination destination, ...)
  {
    ...
  }

  ...

  public Type SourceType { get; }
  public Type DestinationType { get; }
  public bool Equals(TypePair other) =>
    SourceType == other.SourceType
    && DestinationType
        == other.DestinationType;
}
```

(a)

```
struct TypePair : IEquatable<TypePair>
{
  public static TypePair Create<TSource,
      TDestination>(TSource source,
        TDestination scsqbhj, ...)
  {
    ...
  }

  ...

  public Type SourceType { get; }
  public Type DestinationType { get; }
  public bool Equals(TypePair other) =>
    SourceType == other.SourceType
    && DestinationType
        == other.SourceType;
}
```

(b)

Figure 7.6: A C# VarMisuse example which is classified correctly as `DestinationType` in the method `Equals` by the GGNN model of Allamanis et al. (2018). Given the code in Figure 7.6a and the target `SourceType`, our approach renames a local variable `destination` in *another method* to the specific name `scsqbhj`, making the model predict the wrong variable in the method `Equals`, thus ("maliciously") introducing a real bug in the method `Equals`. Additional examples are shown in Yefet et al. (2020).

code2vec model (Alon et al., 2019c) applied to this code snippet predicts the correct name, `sort`, with a probability of 98.54%.

Given the code snippet `f1` and the arbitrarily chosen adversarial target `contains`, our approach finds that renaming the original variable `array` to `ttypes` in `f2` forces the model to predict the label `contains` with an even higher probability, although the functionality remained the same. We denote this as a VarName attack.

Given the code snippet `f1` and the adversarial target `escape`, our approach finds that adding an unused variable (i.e., dead code) named `upperhexdigits` to the end of `f1` forces the model to predict the label `escape` with a probability of 100% (see `f3` in Figure 7.4). We denote this attack as a DeadCode targeted attack.

In general, bypassing semantic labeling models can allow a variety of malicious behaviors. For example, a malicious developer can spread malicious code to users by having the code classified as "benign" by a malware detection model (David et al., 2020; Saxe and Berlin, 2015; Wang et al., 2017; Arp et al., 2014; Grosse et al., 2016). In this work, we demonstrate the ability to bypass semantic labeling by applying our attack on the code2vec model (Figure 7.4), forcing the model to predict a label of our choice.

Bypass Bug Detection (VarMisuse - C#) As another example, we demonstrate how our approach can force a Graph Neural Network (GNN) model to choose an ob-

vious bug as the correct completion. In Figure 7.6a, a GNN model trained on the VarMisuse task (Allamanis et al., 2018; Brockschmidt, 2020) in C# correctly chooses to "fill the blank" using the field `DestinationType` inside the method `Equals`. By renaming a local variable called `destination` in *another method* to the specific name `scsqbhj` (Figure 7.6b), the model chooses the incorrect field `SourceType` in the method `Equals`. The fields `DestinationType` (correct) and `SourceType` (incorrect) both have the same type; thus, the code still compiles and the attack causes a real bug in `Equals`.

More generally, bypassing a bug detection model (Pradel and Sen, 2018; Rice et al., 2017; Bader et al., 2019) can allow a malicious developer inside an organization or inside an open-source project to intentionally introduce bugs. In this work, we demonstrate this ability using the VarMisuse on Graph Neural Networks (GNNs) (Figure 7.6), forcing the model to choose an incorrect (but type-correct) variable.

In addition to the code2vec and VarMisuse tasks that we address in this work, we believe adversarial examples can be applied to neural code search (Sachdev et al., 2018; Liu et al., 2019a; Cambronero et al., 2019). A developer can attract users to a specific library or an open-source project by introducing code that will be disproportionately highly ranked by a neural code search model.

7.2.4 Discrete Adversarial Manipulation of Programs (DAMP)

Consider the code snippet `f1` of Figure 7.4 that sorts a given array. The code2vec model (Alon et al., 2019c) applied to this code snippet predicts the correct name, `sort`. Our goal is to find *semantically equivalent* snippets that will cause an underlying model to yield an incorrect target prediction of our choice.

Gradient-Based Exploration of the Program Space We need a way to guide exploration of the program space towards a specific desired target label (in a targeted attack), or away from the original label (in a non-targeted attack).

In standard stochastic gradient descent (SGD)-based training of neural networks, the weights of the network are updated to minimize the loss function. The gradient is used to guide the update of the network weights to minimize the loss. However, what we are trying to determine is not an update of the network's weights, but rather an "update" of the network's *inputs*. A natural way to obtain such guidance is to derive the desired prediction with respect to the model's *inputs* while holding the model weights constant and follow the gradient to modify the inputs.

In settings where the input is continuous (e.g., images), modifying the input can be done directly by adding a small noise value and following the direction of the gradient towards the desired target label (targeted), or away from the original label (non-targeted). A common technique used for images is the *Fast Gradient Signed Method* (FGSM) (Goodfellow et al., 2014b) approach, which modifies the input using a small fixed ϵ value.

Deriving with Respect to a Discrete Input In settings where the input is discrete, the first layer of a neural network is typically an embedding layer that embeds discrete objects, such as names and tokens, into a continuous space (Alon et al., 2019a; Allamanis et al., 2016; Iyer et al., 2016). The input is the index of the symbol, which is used to look up its embedding in the embedding matrix. The question for discrete inputs is therefore: *what does it mean to derive with respect to the model's inputs?*

One approach is to derive with respect to the *embedding vector*, which is the result of the embedding layer. In this approach, after the gradient is obtained, we need to reflect the update of the embedding vector back to discrete-input space. This can be done by looking for the nearest-neighbors of the updated embedding vector in the original embedding space, and finding a nearby vector that has a corresponding discrete input. In this approach, there is no guarantee that following the gradient is the best step.

In contrast, our Discrete Adversarial Manipulation of Programs (DAMP) approach derives with respect to a one-hot vector that represents the *distribution* over discrete values (e.g., over variable names). Instead of deriving by the input itself, the gradient is taken with respect to the *distribution* over the inputs. Intuitively, this allows us to directly obtain the best discrete value for following the gradient.

Targeted Gradient-based Attack Using our gradient-based method, we explore the space of semantically equivalent programs *directly toward* a desired adversarial target. For example, given the code snippet `f1` of Figure 7.4 and the desired target label `contains`, our approach for generating adversarial examples automatically infers the snippet `f2` of Figure 7.4. Similarly, given the target label `escape`, our approach automatically infers the snippet `f3` of Figure 7.4.

All code snippets of Figure 7.4 are semantically equivalent. The only difference between `f1` and `f2` is the name of the variables. Specifically, these snippets differ only in the name of a single variable, which is named `array` in `f1` and `ttypes` in `f2`. Nevertheless, when `array` is renamed to `ttypes`, the prediction made by CODE2VEC changes to the desired (adversarial) target label `contains`. The difference between `f1` and `f3` is the addition of a single variable declaration `int upperhexdigits`, which is never used in the code snippet. Nevertheless, adding this declaration changes the prediction made by the model to the desired (adversarial) target label `escape`.

More details can be found in Yefet et al. (2020). In Finkelshtein et al. (2020), we extend adversarial attacks on programs to general graphs.

Chapter 8

Conclusion

In this dissertation, we presented a simple and general approach for learning from programs. The main idea is to represent a program using paths in its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of source code rather than treating it as a flat sequence of tokens.

We showed that this representation can serve as basis for models that are trained on up to $16M$ methods examples, and be useful for a variety of programming languages and prediction tasks: predicting variable names, predicting method names, generating a natural language sequence given a code snippet, and the most challenging task: any-code completion.

We showed that these models are easily generalizable to different programming languages, including JavaScript, Java, Python, and C#. Our models perform significantly better than previous programming-language-oriented works and state-of-the-art NMT models applied in our settings. Our approaches generalize many previous work in this area while reaching state-of-the-art performance on challenging benchmarks.

While comparing our approach and existing methods, we found theoretical explanations to the empirical differences between different models. In particular, we found a novel explanation to a well known limitation in training graph neural networks: a bottleneck that causes over-squashing. As a result, GNNs fails to propagate long-range information, learn only short-range signals from the training data instead, and performs poorly when the prediction task depends on long-range interaction.

We believe that the principles presented in this thesis can serve as a basis for a wide range of tasks which involve source code, source code and natural language, and can be extended to other kinds of generated outputs. Since the representation of programs using AST paths is fundamental to programming languages, it can be used in a variety of other machine learning tasks, including different applications and different learning models. We also believe that structural language modeling enables a wide range of future applications, similarly to how language modeling research has contributed to NLP in recent years.

Code, data, and trained models can be found at:

```
https://github.com/tech-srl/PigeonJS/
https://github.com/tech-srl/code2vec/
https://github.com/tech-srl/code2seq/
https://github.com/tech-srl/slm-code-generation/
https://github.com/tech-srl/bottleneck/
https://github.com/tech-srl/how_attentive_are_gats/
```

Bibliography

JavaParser. http://javaparser.org.

Roslyn. https://github.com/dotnet/roslyn.

UglifyJS. https://github.com/mishoo/UglifyJS.

UnuglifyJS. https://github.com/eth-srl/UnuglifyJS.

Roee Aharoni and Yoav Goldberg. Towards string-to-tree neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 132–140, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-2021. URL https://www.aclweb.org/anthology/P17-2021.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. *arXiv preprint arXiv:1812.06469*, 2018b.

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487127.

Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635901. URL http://doi.acm.org/10.1145/2635868.2635901.

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635883. URL http://doi.acm.org/10.1145/2635868.2635883.

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on*

Foundations of Software Engineering, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *JMLR Proceedings*, pages 2123–2132. JMLR.org, 2015b.

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2091–2100, 2016. URL http://jmlr.org/proceedings/papers/v48/allamanis16.html.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182*, 2017.

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 and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=i80OPhOCVH2.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192412. URL http://doi.acm.org/10.1145/3192366.3192412.

Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=H1gKYo09tX.

Uri Alon, Golan Pundak, and Tara N Sainath. Contextual speech recognition with difficult negative training examples. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6440–6444. IEEE, 2019b.

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 2019c. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.

Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

Christof Angermueller, Tanel Pärnamaa, Leopold Parts, and Oliver Stegle. Deep learning for computational biology. *Molecular systems biology*, 12(7):878, 2016.

Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in neural information processing systems*, pages 1993–2001, 2016.

Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–27, 2019.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 4945–4949. IEEE, 2016.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=944919.944966`.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2933–2942, 2016. URL `http://jmlr.org/proceedings/papers/v48/bielik16.html`.

Pavol Bielik, Veselin Raychev, and Martin Vechev. Program synthesis for character level language modeling. In *ICLR*, 2017.

Marc Brockschmidt. Gnn-film: Graph neural networks with feature-wise linear modulation. *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2020.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Bke4KsA5FX`.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021.

S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35, Oct 2009. doi: 10.1109/WCRE.2009.50.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974, 2019.

Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2018.

Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the Thirty-Fourth Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2020a.

Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *International Conference on Machine Learning*, pages 942–950, 2018a.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, pages 1725–1735. PMLR, 2020b.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, pages 2547–2557, 2018b.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019. URL `http://arxiv.org/abs/1901.01808`.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, 2014a.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014b.

Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, pages 577–585, 2015.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390177. URL `http://doi.acm.org/10.1145/1390156.1390177`.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 243–253, 2020.

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pages 990–998, 2017.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SJeqs6EFvB`.

Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, 2018.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751*, 2017.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 9165–9174. Curran Associates, Inc., 2019. URL `http://papers.nips.cc/paper/9116-write-execute-assess-program-synthesis-with-a-repl.pdf`.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547, 2020.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1ersoRqtm`.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Ben Finkelshtein, Chaim Baskin, Evgenii Zheltonozhskii, and Uri Alon. Single-node attack for fooling graph neural networks. *arXiv preprint arXiv:2011.03574*, 2020.

Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

Shogo Fujita, Hidetaka Kamigaito, Hiroya Takamura, and Manabu Okumura. Pointing to subwords for generating function names in source code. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 316–327, 2020.

Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1213–1222. JMLR. org, 2017.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 513–520, 2011.

Jonathan Godwin, Michael Schaarschmidt, Alexander Gaunt, Alvaro Sanchez-Gonzalez, Yulia Rubanova, Petar Veličković, James Kirkpatrick, and Peter Battaglia. Very deep graph neural networks via noise regularisation. *arXiv preprint arXiv:2106.07971*, 2021.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014a.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014b.

Aleksa Gordić. pytorch-gat. `https://github.com/gordicaleksa/pytorch-GAT`, 2020.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE, 2005.

Cordell Green. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier, 1981.

Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*, 2016.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, volume 46, pages 317–330. ACM, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

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. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=jLoC4ez43PZ`.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337322`.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al.

Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_14. URL http://dx.doi.org/10.1007/978-3-642-03013-0_14.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

Binxuan Huang and Kathleen M Carley. Syntax-aware aspect level sentiment classification with graph attention networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5472–5480, 2019.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016. URL http://aclweb.org/anthology/P/P16/P16-1195.pdf.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086*, 2019.

Chaitanya Joshi. Transformers are graph neural networks. *The Gradient*, 2020.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE, 2019.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. *ArXiv e-prints*, 2017.

Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *International Conference on Learning Representations*, 2018.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1557769.1557821`.

Vineet Kosaraju, Amir Sadeghian, Roberto Martín-Martín, Ian Reid, Hamid Rezatofighi, and Silvio Savarese. Social-bigat: Multimodal trajectory forecasting using bicycle-gan and graph attention networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/d09bf41544a3365a46c9077ebb5e35c3-Paper.pdf`.

Devin Kreuzer, Dominique Beaini, William L Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. *arXiv preprint arXiv:2106.03893*, 2021.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

Vitaly Kurin, Maximilian Igl, Tim Rocktäschel, Wendelin Boehmer, and Shimon Whiteson. My body is a cage: the role of morphology in graph-based incompatible control. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=N3zUDGN5lO.

Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Tony Jebara and Eric P. Xing, editors, *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196. JMLR Workshop and Conference Proceedings, 2014. URL http://jmlr.org/proceedings/papers/v32/le14.pdf.

Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *ACL (2)*, pages 302–308. Citeseer, 2014a.

Omer Levy and Yoav Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the eighteenth conference on computational natural language learning*, pages 171–180, 2014b.

Omer Levy, Minjoon Seo, Eunsol Choi, and Luke Zettlemoyer. Zero-shot relation extraction via reading comprehension. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017*, pages 333–342, 2017. doi: 10.18653/v1/K17-1034. URL https://doi.org/10.18653/v1/K17-1034.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

Ben Liblit, Andrew Begel, and Eve Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, Sussex, United Kingdom, sep 2006. Psychology of Programming Interest Group.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation.

In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1057. URL https://www.aclweb.org/anthology/P16-1057.

Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. Neural query expansion for code search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 29–37, New York, NY, USA, 2019a. ACM. ISBN 978-1-4503-6719-6. doi: 10.1145/3315508.3329975. URL http://doi.acm.org/10.1145/3315508.3329975.

Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12. IEEE, 2019b.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*, 2021.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, October 2017. ISSN 2475-1421. doi: 10.1145/3133908. URL http://doi.acm.org/10.1145/3133908.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-2045. URL http://www.aclweb.org/anthology/P17-2045.

Denis Lukovnikov and Asja Fischer. Gated relational graph attention networks, 2021. URL https://openreview.net/forum?id=v-9E8egy_i.

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015. URL http://aclweb.org/anthology/D/D15/D15-1166.pdf.

Nianzu Ma, Sahisnu Mazumder, Hao Wang, and Bing Liu. Entity-aware dependency-based deep graph attention network for comparative preference classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5782–5788, 2020.

Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning - Volume 32*, ICML'14, pages II–649–II–657. JMLR.org, 2014. URL http://dl.acm.org/citation.cfm?id=3044805.3044965.

Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013a. URL http://arxiv.org/abs/1301.3781.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 3111–3119, USA, 2013b. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999792.2999959.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. 2013c.

Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384689. URL http://doi.acm.org/10.1145/2384616.2384689.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 2204–2212, Cambridge, MA, USA, 2014. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969033.2969073.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5115–5124, 2017.

Christopher Morris, Matthias Fey, and Nils M Kriege. The power of the weisfeiler-leman algorithm for machine learning with graphs. *arXiv preprint arXiv:2105.05911*, 2021.

Dana Movshovitz-Attias and William W Cohen. Natural language models for predicting programming comments. 2013.

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2018. URL http://arxiv.org/abs/1703.05698.

Deepak Nathani, Jatin Chauhan, Charu Sharma, and Manohar Kaul. Learning attention-based embeddings for relation prediction in knowledge graphs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4710–4723, 2019.

Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/k^ 2). In *Dokl. Akad. Nauk SSSR*, volume 269, pages 543–547, 1983.

Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=S1ldO2EFPr`.

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *ICLR*, 2017.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *to appear in ICML'2021*, 2021.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 107–126. ACM, 2015.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, October 2018. ISSN 2475-1421. doi: 10.1145/3276517. URL `http://doi.acm.org/10.1145/3276517`.

Danish Pruthi, Bhuwan Dhingra, and Zachary C Lipton. Combating adversarial misspellings with robust word recognition. *ACL*, 2019.

Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. Deepinf: Social influence prediction with deep learning. In *Proceedings of the 24th*

ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18), 2018.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1105. URL http://www.aclweb.org/anthology/P17-1105.

Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole Von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1:140022, 2014.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594321. URL http://doi.acm.org/10.1145/2666356.2594321.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009. URL http://doi.acm.org/10.1145/2676726.2677009.

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 731–747, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984041. URL http://doi.acm.org/10.1145/2983990.2984041.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA, 2016b. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671. URL http://doi.acm.org/10.1145/2837614.2837671.

Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):104, 2017.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=Hkx1qkrKPr.

Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, 1(2):127–190, 1999.

Reuven Y Rubinstein. Combinatorial optimization, cross-entropy, ants and rare events. *Stochastic Optimization: Algorithms and Applications*, 54:303–363, 2001.

Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 379–389, 2015. URL `http://aclweb.org/anthology/D/D15/D15-1044.pdf`.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 31–41, 2018. doi: 10.1145/3211346.3211353. URL `http://doi.acm.org/10.1145/3211346.3211353`.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020.

Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3): 93–93, 2008.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725,

Berlin, Germany, August 2016. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P16-1162.

Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014. ISBN 1107057132, 9781107057135.

Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *Relational Representation Learning Workshop, NeurIPS 2018*, 2018.

Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Syl8Sn0cK7.

Richard Socher, Cliff C. Lin, Andrew Y. Ng, and Christopher D. Manning. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2011.

Tim Sonnekalb. Machine-learning supported vulnerability detection in source code. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1180–1183, 2019.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566. Association for Computational Linguistics, 2015. doi: 10.3115/v1/P15-1150. URL http://aclweb.org/anthology/P15-1150.

Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996. URL `http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html`.

Rohan Taori, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. Targeted adversarial examples for black box audio systems. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 15–20. IEEE, 2019.

Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3), 2006.

Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 384–394, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1858681.1858721`.

Peter D Turney. Similarity of semantic relations. *Computational Linguistics*, 32(3): 379–416, 2006.

Diego Valsesia, Giulia Fracastoro, and Enrico Magli. Don't stack layers in graph neural networks, wire them randomly. In *ICLR 2021 Workshop on Geometrical and Topological Representation Learning*, 2021.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ByloJ20qtm`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

Martin T. Vechev and Eran Yahav. Programming with "big code". *Foundations and Trends in Programming Languages*, 3(4):231–284, 2016. doi: 10.1561/2500000028. URL `https://doi.org/10.1561/2500000028`.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2015.

Richard J Waldinger and Richard CT Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252. Morgan Kaufmann Publishers Inc., 1969.

Eric Wallace, Mitchell Stern, and Dawn Song. Imitation attacks and defenses for black-box machine translation systems. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5531–5546, 2020.

Guangtao Wang, Rex Ying, Jing Huang, and Jure Leskovec. Improving graph attention networks with large margin-based constraints. *arXiv preprint arXiv:1910.11945*, 2019a.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019b.

Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia II, Xinyu Xing, Xue Liu, and C Lee Giles. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1145–1153. ACM, 2017.

Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, pages 2022–2032, 2019c.

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-0-7695-5594-2. URL `http://dl.acm.org/citation.cfm?id=2820518.2820559`.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1341–1350, 2016.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Yiding Yang, Jiayan Qiu, Mingli Song, Dacheng Tao, and Xinchao Wang. Distilling knowledge from graph convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=v9hAX77--cZ.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1041. URL http://www.aclweb.org/anthology/P17-1041.

Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, 2018.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJl6AjC5F7.

Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7144–7153, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/young19a.html.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. TypeSQL: Knowledge-based type-aware neural text-to-SQL generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2093. URL https://www.aclweb.org/anthology/N18-2093.

Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. Towards context-aware code comment generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 3938–3947, 2020.

Kai Zhang, Yaokang Zhu, Jun Wang, and Jie Zhang. Adaptive structural fingerprints for graph attention networks. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJxWx0NYPr`.

Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=rkecl1rtwB`.

Rui Zhao, David Bieber, Kevin Swersky, and Daniel Tarlow. Neural networks for modeling source code edits, 2019.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=Xh5eMZVONGF`.

ומודלים המבוססים על רשתות נוירונים גרפיות, מצאנו כי רשתות הנוירונים הגרפיות מתקשות ללמוד תבניות ארוכות-טווח בדאטא. כלומר, אם החיזוי ה"נכון" תלוי בתבניות הכוללות צמתים המרוחקים אחד מהשני בגרף, הרשת לא מצליחה ללמוד אותן, ובמקום זה, הרשת מתאמנת-יתר על תבניות קצרות-טווח. תופעה זו הייתה מפתיעה, מכיוון שמסלולי AST בדרך כלל אינם מתקשים כלל בללמוד תבניות ארוכות-טווח. בסקירה ספרותית, מצאנו כי מאז המצאת רשתות הנוירונים הגרפיות בשנת 2005, הקושי שלהן בללמוד תבניות ארוכות-טווח נצפה והיה ידוע כבעיה באימון רשתות אלו. אולם, בעיה זו מעולם לא הוסברה ונחקרה כראוי. בתזה זאת, אנחנו מציגים את בעיית "צוואר הבקבוק" של רשתות הנוירונים הגרפיות, מציעים הסבר חדש לתופעה זו, מראים שתופעה זו מתקיימת אף במודלים קיימים, וכי סוגים שונים של רשתות נוירונים גרפיות - נפגעות מ"צוואר הבקבוק" באופן שונה. תגלית זו שופכת אור ומשלימה את התמונה אודות ייצוגים פופולריים שונים של תוכניות, חולשותיהם וחוזקותיהם.

תקציר

במהלך העשור האחרון, מהפכת הלמידה העמוקה, הנשענת על רשתות נוירונים מלאכותיות, שינתה ללא היכר טווח רחב של תחומים במדעי המחשב כגון ראייה ממוחשבת, זיהוי דיבור, ועיבוד שפה טבעית. במקביל, גדל דרמטית מספר בסיסי הקוד ופרויקטי הקוד הפתוח הזמינים לציבור, מה שאפשר את השימוש ברשתות נוירונים במגוון רחב של משימות הקשורות לתכנות וכתיבת קוד, תחום שאנחנו מכנים "עיבוד שפות תכנות" (Programming Language Processing – PLP, כאנלוגיה ל"עיבוד שפה טבעית" – NLP).

עם זאת, בעיית הייצוג של תוכניות מחשב באלגוריתמים לומדים ובמערכות למידה עמוקה נותרה כשאלה פתוחה. ברור כי לתוכניות אין ייצוג "מטריציוני" ישיר ופשוט כמו שיש, למשל, לתמונות. אף על פי שתוכנית מחשב יכולה להיות מיוצגת כרצף של "מילים" או רצף של תווים, כמו טקסט בשפה טבעית, תוכניות מחשב הן הרבה יותר מבניות מטקסט-חופשי, מכיוון שתוכניות חייבות לציית לתחביר נוקשה ועשיר המוגדר על ידי דקדוק חסר-הקשר. יתרה מזאת, לכל שפת תכנות יש סמנטיקה מוגדרת מראש, המתארת מה פירושן של תוכניות תקינות-דקדוקית ומה הן עושות.

תזה זו מתמקדת בבעיה הכללית הבאה: ייצוג תוכניות מחשב באלגוריתמים לומדים ומערכות למידה עמוקה באופן שמקל על הלמידה, בעודו תופס מידע רב ככל הניתן מתוכניות הקלט, וגם מאפשר למודל הלומד להשאר כללי ככל האפשר. תזה זו מציגה את גישת מסלולי ה-AST, אשר מייצגת תוכניות באמצעות מסלולים בעץ התחביר המופשט (Abstract Syntax Tree – AST) של התוכנית. ייצוג זה, מסלולי ה-AST, מאפשר לבנות מודלים נוירוניים חזקים ומדויקים, אבל עם זאת גם קלי-משקל וניתנים לאימון על כמות מידע מאסיבית. ספציפית, תזה זאת מראה איך ניתן לאמן את המודלים הללו על מיליוני דוגמאות, עבור משימות הכוללות חיזוי תכונות של רכיבי תוכנית בודדים, חיזוי תכונות של קטעי קוד שלמים, חיזוי של משפט בשפה טבעית מתוך קטע קוד נתון, והפקה של השלמות קוד אוטומטיות. מודלים אלו פורסמו בפומבי כאתרי אינטרנט של הדגמות אינטראקטיביות, לצד קוד-פתוח של מימושים ודאטא. חלק מהמודלים הללו, כמו CODE2VEC ו-CODE2SEQ, הינם פופולריים במיוחד, ונמצאים בשימוש נרחב באקדמיה ובתעשייה.

לבסוף, תזה זאת חוקרת את ההבדלים התיאורטיים בין ייצוגים שונים של תוכניות. ספציפית, המחקר שהתחיל כממוקד בתוכניות הוביל לגילוי מגבלות אינהרנטיות רחבות יותר הקיימות בייצוג פופולרי אחר, רשתות הנוירונים הגרפיות (GNN) . רשתות הנוירונים הגרפיות נעשו פופולריות ביותר בשלוש השנים האחרונות, עקב הכלליות והרב-גוניות שלהן. למעשה, כל דאטא שניתן להצגה כגרף, כגון עץ התחביר של תוכנית מחשב, רשת חברתית, ומולקולה - ניתן בקלות להזין לתוך רשת נוירונים גרפית. זמן האימון של רשת נוירונים גרפית הוא יחסית מהיר, ואסימפטוטית הוא תלוי לינארית במספר הצמתים ומספר הקשתות. כאשר השווינו אמפירית את המודלים המבוססים על מסלולי AST

המחקר בוצע בהנחייתו של פרופסור ערן יהב, בפקולטה למדעי המחשב.

חיבור זה מבוסס על הפרסומים העיקריים הבאים:

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=i80OPhOCVH2`.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192412. URL `http://doi.acm.org/10.1145/3192366.3192412`.

Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL `https://openreview.net/forum?id=H1gKYo09tX`.

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`.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.

הפרסומים הבאים היו חלק מהמחקר שבוצעו במהלך תקופת הדוקטורט של המחבר, והם מציגים תוצאות משלימות לחיבור זה:

Uri Alon, Golan Pundak, and Tara N Sainath. Contextual speech recognition with difficult negative training examples. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6440–6444. IEEE, 2019.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

תודות

אני מודה מעומק לבי על המימון הנדיב של מחקר זה לטכניון, לפקולטה למדעי המחשב על-שם הנרי ומרילין טאוב, ולמלגת אירווין וג'ואן ג'ייקובס.

עיבוד שפות תכנות באמצעות למידת מכונה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אורי אלון

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
אב התשפ"א חיפה יוני 2021

עיבוד שפות תכנות באמצעות למידת מכונה

אורי אלון